Universität Leipzig

Fakultät für Mathematik und Informatik

(Institut für Informatik)

Entwurf und Implementation einer echtzeitfähigen Entwicklungsumgebung für Lern- und Evolutionsexperimente mit autonomen Robotern.

Diplomarbeit

vorgelegt von: Thomas Pantzer

betreut von: Prof. Ralf Der, Universität Leipzig

Leipzig, September 2000

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, um all jenen meinen herzlichen Dank auszusprechen, die mir den Abschluß dieser Arbeit ermöglicht haben. Zuallererst möchte ich meinem Betreuer Ralf Der danken, der mir immer mit gutem Rat zur Seite stand. Er verstand es auch, mir in den richtigen Momenten wieder Motivation für den nächsten zu bewältigenden Schritt zu geben. Die Rechte der deutschen Sprache verdanken ihre Durchsetzung in nicht unerheblichen Maße Fabian Schmidt, Dietmar Fischer und Lydia Thiessen, denen dafür von Herzen gedankt sei. Meinen Eltern sei gedankt, daß sie mir nicht zuletzt auch finanziell den Rücken freigehalten haben, damit ich mich ganz meiner Aufgabe widmen konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Ein in unbekannter Umgebung agierender autonomer Roboter muß Probleme der Positionsbestimmung, Navigation, Rückkehr zur Basisstation etc. sicher lösen können, um seine eigentliche Aufgabe durchführen zu können. In der Vergangenheit wurden verschiedene Ansätze entwickelt, um z.B. die Positionsbestimmung [HPG97][SG98] und Navigation [MLP+98][MM80] zu verbessern oder um effiziente Controller zu entwickeln. Dabei zeigte sich, daß viele Verfahren, die mit Computersimulation entwickelt wurden, nur nach starken Anpassungen für eine bestimmte Hardware ausreichend gut funktionierten. Die bei den meist vereinfachten Computermodellen weggelassenen Eigenschaften der Hardware und die Differenzen von simulierter und realer Welt verursachen oft unvorhergesehene Nebeneffekte. Deshalb ist es notwendig, die neuen Algorithmen und Verfahren auch an "richtiger" Hardware zu erproben. Im folgenden wird eine Entwicklungsumgebung vorgestellt, mit der neue Algorithmen an konkreter Hardware getestet werden können.

Kapitel 1

Einleitung

1.1 Motivation

Ein autonomer mobiler Roboter soll in einer realen Welt den Menschen von monotonen, immer wiederkehrenden Arbeiten ebenso entlasten wie von Tätigkeiten, die mit Gefahr für Leib und Leben verbunden sind. Mit fortschreitender technischer Entwicklung werden auch immer entlegenere oder dem Menschen unzugängliche Gebiete zu Forschungsgegenständen. Dabei werden nicht nur technologische sondern auch an prinzipielle oder durch Naturgesetze vorgegebene Grenzen erreicht. So wurde zum Beispiel die Marssonde Pathfinder von der Erde aus ferngesteuert. Eine mühselige Angelegenheit, die Steuer- und Bildsignale (mit Lichtgeschwindigkeit übertragen) benötigen ca. 11 Minuten, um vom Mars zur Erde zu gelangen. Bei dieser Verzögerung ist es nahezu unmöglich, auf ein unvorhergesehenes Hindernis zu reagieren. Auf den Einsatz autonomer mobiler Roboter wurde aus Sicherheits- und Kostengründen verzichtet. Die Weiterentwicklung autonom agierender intelligenter Agenten als Basis für solche und ähnliche Forschungsvorhaben sind wichtige Bausteine auf die in Zukunft nicht mehr verzichtet werden wird.

Zur Entwicklung autonomer Agenten werden verschiedene Ansätze verfolgt. Künstliche Evolution [Flo99a], *reinforcement learning* [DH95], fallbasiertes Schließen oder das Prinzip der *homeokinesis* [DSP99] seien an dieser Stelle genannt. Im allgemeinen werden diese Strategien auf dem Computer modelliert und mit Simulationen getestet. Werden die Modelle dabei zu allgemein gewählt, kann es passieren, daß in der Simulation gut funktionierende Algorithmen nur teilweise, schlecht oder überhaupt nicht in Implementationen mit konkreter Hardware funktionieren. So kann zum Beispiel die

Annahme, daß die Veränderung der Sensordaten auf eine kontinuierliche Funktion in der Zeit abbildbar ist, erhebliche Seiteneffekte hervorrufen. Im konkreten Beispiel der Marssonde Pathfinder wäre die Zeitdauer für einen Koordinationsschritt Sensor/Motor ca. 22 Minuten lang. Die Annahme gilt also nur noch, wenn sich der Roboter sehr langsam bewegt. Das ist aber nicht beliebig ausdehnbar – irgendwann ist jede Batterie erschöpft.

Als zweites Beispiel kann die Annahme genannt werden, daß Aktuatoren mit jedem beliebigen Wert aus dem Bereich der reellen Zahlen angesteuert werden können. Meistens ist es aber der Fall, daß die Hardware, bedingt durch die Konstruktion, die übergebenen Daten auf Vielfache einer kleinsten Einheit trunkiert. Ist dieser Wert nicht hinreichend klein, sind Interferenz- oder Resonanzeffekte spürbar, die die Funktion des Roboters beeinträchtigen oder sogar die Hardware des Roboters beschädigen können. Der in der Simulation getestete Algorithmus kann auf dem realen Roboter auch wegen Fertigungstoleranzen der verwendeten Sensoren oftmals seine Stärken nicht voll ausspielen. In [NML95] werden Methoden aufgezeigt, wie Sensoren in Simulationen modelliert werden können, um die Lücke zwischen Simulation und realer Welt nicht zu groß werden zu lassen.

In [PS98] beschreibt R. Pfeiffer mit seinem Ansatz der *embodied intelligence* weitere wichtige Gründe, warum es sinnvoll ist, reale Robotermodelle für Forschungszwecke einzusetzen.

> "Ein autonomer mobiler Agent in der realen Welt ist einem hochdimensionalen, sich kontinuierlich ändernden Strom von Sensordaten ausgesetzt. Wenn er Klassifizierungen dieser Daten vornehmen soll, muß er die Dimensionen dieses Datenstromes reduzieren können. Der Ansatz mit Lernen ist relativ schwierig zu realisieren, weil die Invarianzen nicht direkt im Datenstrom zu erkennen sind. Die Betrachtungsweise der *embodied cognitive science* bevorzugt einen anderen Ansatz, der auf die *Natur* der Daten eingeht: Der Agent wird nicht passiv dem Datenstrom ausgesetzt, sondern mit einem Körper ausgestattet, der ihn in die Lage versetzt, die Daten selbst durch Interaktion mit der Umwelt zu erzeugen. Das heißt, er kann korrelierte Datensätze erzeugen, die einfacher zu lernen sind, weil sie Redundanzen der jeweiligen Interaktionen enthalten. Damit wird die Komplexität des Lernens um Größenordnungen kleiner."

Eine Testumgebung für einen Prototyp eines Roboters zu bauen und mit realen Robotermodellen zu arbeiten, ist also notwendig und sinnvoll. Andererseits gibt es bei die-

sem Ansatz auch Nachteile, die eine genaue Abwägung erfordern, ob und wann eine Simulation zu den gleichen Resultaten führen kann. Als Hauptaspekt ist dabei die lange Gesamtdurchführungszeit für ein Experiment zu nennen, die durch die physikalischen Gegebenheiten bedingt ist. Daten müssen transferiert, Aktuatoren wirklich bewegt werden. Dabei entstehen Verzögerungen, die in einfachen Simulatoren entweder nicht berücksichtigt oder durch einfaches Setzen von Variablen implementiert werden. Diese Verzögerungen sind u.a. auf die Masseträgheit und Motorleistung zurückzuführen und können Werte im Sekundenbereich annehmen. Bei der Anwendung von Simulationen beschränkt sich dieser Wert auf die Ausführung von Maschinenbefehlen und liegt typischerweise im Mikrosekundenbereich.

Ein weiterer großer Nachteil ist die fehlende Parallelität. Auf einem schnellen Rechner können mehrere Simulationsprozesse gleichzeitig gestartet werden, für den Versuch mit realen Robotermodellen wird mehrfach dieselbe Hardware benötigt, oder es wird z.B. bei der künstlichen Evolution jeder Instanz einer Population nacheinander die Hardwareressource "Roboter" zugewiesen.

Unabhängig davon ist die Entwicklung und Evolution von Algorithmen auf realen Robotern *in vivo* – also in Echtzeit – eine Herausforderung, die nur von wenigen Wissenschaftlern [NML95],[NF97],[Flo99b] des Forschungsgebietes *Autonome Roboter* bisher in Angriff genommen wurde.

In dieser Arbeit wird das Grundgerüst für eine Software vorgestellt, die die Brücke vom Algorithmus zur realen Hardware herstellt. Beim Design dieser Software konnte auf Erfahrungen, die durch das Experimentieren mit dem *khepera*-Roboter gewonnen wurden, zurückgegriffen werden.

1.2 Andere Arbeiten

Oliver Michel realisierte als erster einen khepera-Simulator der den Roboter auch über die serielle Schnittstelle steuern kann. Dieser Simulator ist am weitesten verbreitet, inzwischen existiert auch eine 3D-Version. Der Simulator kann nur die Basisfunktionen des *khepera*-Roboters ansteuern. Da diesem Simulator kein objektorientiertes Design zugrunde liegt, ist der Aufwand beim Programmieren relativ hoch.

Erol Sahin [SG99] und Paolo Gaudiano realisierten eine Entwicklungsumgebung für *khepera*-Roboter zur Evaluierung von Positionsbestimmung und Navigationsalgorithmen. Es können die vom Roboter berechneten Positionsdaten mit den von einer über dem Versuchsfeld montierten Kamera erfaßten realen Positionsdaten in Echtzeit verglichen werden. Das System enthält umfangreiche Algorithmen zur Bildverarbeitung (Segmentierer), mit denen farbige Objekte erkannt werden können. Positionsinformationen können auch aus der sich ändernden Größe im Bild bei sich bewegenden Objekten abgeleitet werden [SG98]. Die Software ist unter der GPL[1] frei verfügbar.

In [LPJ+96] wird die Meteaur-Plattform zur Modellierung und Erprobung von Algorithmen für autonome Roboter beschrieben. Die verwendeten Software-Werkzeuge sind nicht öffentlich verfügbar.

[1] Gnu Public License

Kapitel 2

Grundlagen

2.1 Begriffe

2.1.1 Algorithmus, Agent, Controller

Ein mathematischer Algorithmus in seiner konkreten Implementation als Objekt in der Steuerungssoftware wird als Agent bezeichnet. Wenn dieser Agent nicht nur Sensorwerte einliest, sondern auch Aktuatoren des Roboters steuert, wird er zum Controller. Ein Controller verarbeitet Eingabesignale zu Ausgabesignalen und stellt einen Regelkreis her. In einigen Fällen wird auch von der CPU des Roboters einschließlich der darin implementierten Routinen als Controller bzw. Roboter-Controller gesprochen.

2.1.2 Echtzeit

Unter Echtzeitfähigkeit eines Systems oder Computers wird die Fähigkeit verstanden, innerhalb einer definierten Zeittoleranz auf ein Ereignis zu reagieren. Dabei wird zwischen weichen und harten Echtzeitanforderungen unterschieden. Eine weiche Echtzeitanforderung ist z.B. das Bewegen des Mauszeigers auf dem Bildschirm. Eine Verzögerung wird vom Nutzer hingenommen. Eine harte Echtzeitanforderung ist das Brennen einer CD. Werden die Daten nicht kontinuierlich geliefert (konstante Daten-

menge pro Zeitfenster), wird der Rohling unbrauchbar. Bei Robotikexperimenten bestehen in vielen Fällen harte Echtzeitanforderungen. Trotzdem wurde auf den Einsatz eines Echtzeitbetriebssystems auf dem Hostrechner verzichtet. Auf die Aspekte des Softwaredesigns unter einem speziellen Echtzeitbetriebssystem wird hier nicht eingegangen. Die verwendeten Computer sind i.a. schnell genug, um die Echtzeitanforderungen in den betrachteten Fällen zu erfüllen.

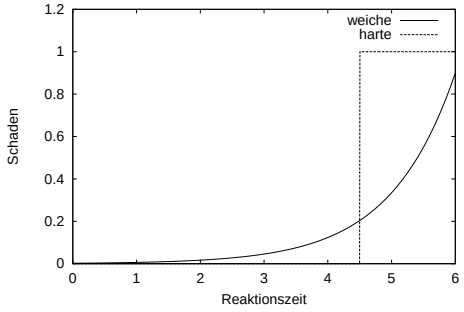


Abbildung 2.1: Schadensverlauf bei Echtzeitanforderungen

2.1.3 Zeitschritt, Updateschritt

Ein Zeitschritt ist die Zeitspanne zwischen zwei Sensorwertmessungen. Je kürzer ein Zeitschritt ist, um so höher ist die Genauigkeit der aufgenommenen Meßkurve oder die Feinheit der Steuerung (siehe Punkt 2.3.1).

2.1.4 Blockierende vs. nicht-blockierende Ein-/Ausgabe

Das UNIX-Betriebssystem stellt zur Programmierung der Ein- und Ausgabe die Funktion *read(...)* zur Verfügung. Wird ein Dateideskriptor standardmäßig geöffnet, blockiert diese Funktion beim Lesen aus diesem Deskriptor den Programmablauf solange, bis Daten bereit sind. Beim Schreiben blockiert die Funktion *write(...)* so lang, bis alle Daten auf die Hardware geschrieben wurden. Wurde der Dateideskriptor im Modus `O_NONBLOCK` geöffnet, wird nach *read(...)* und *write(...)* die Programmabarbeitung sofort wieder aufgenommen.

2.1.5 Trajektorie

Eine Trajektorie ist eine Menge von n-dimensionalen Vektoren, die nach dem Zeitpunkt ihrer Erzeugung geordnet sind. So ist z. B. eine Trajektorie im Sensorraum die Abfolge der einzelnen Sensorwerte pro Zeitschritt, die beim Vorbeifahren an einem Objekt aufgenommen wurden. Die Dimension des Raumes entspricht dabei der Anzahl der betrachteten Sensoren. Die Abfolge der Positionsinformationen (x/y-Koordinaten) des Roboters ist ebenfalls eine Trajektorie und bezeichnet den zurückgelegten Weg des Roboters im Weltkoordinatensystem.

2.1.6 Odometrie

Bei der Odometrie, in der Literatur auch als Koppelnavigation oder engl. "dead reckoning" bezeichnet, werden Signalgeber an den Rädern benutzt, um den zurückgelegten Weg des Fahrzeugs (Roboter) messen zu können. Die Odometrie ist eine relative Positionsbestimmung. Die neue Position wird dabei aus einer vorher bekannten Position plus der zurückgelegten Wegstrecke errechnet. Die Odometrie kann auf kurzen Distanzen sehr genaue Ergebnisse liefern. Allerdings entstehen auch Fehler, die mit zunehmender Entfernung wachsen. Die Räder können rutschen (Schlupf), unrund sein oder unterschiedliche Durchmesser haben (Reifenluftdruck), oder es können durch Bodenunebenheiten falsche Entfernungen gemessen werden. In der Literatur [Gut96], [GHH+98] und [KS97] sind weitere Ansätze zur Selbstlokalisation ausführlich beschrieben.

2.1.6.1 Formeln

In Abbildung 2.2 ist die Positionsveränderung eines zweirädrigen Fahrzeugs im Koor-

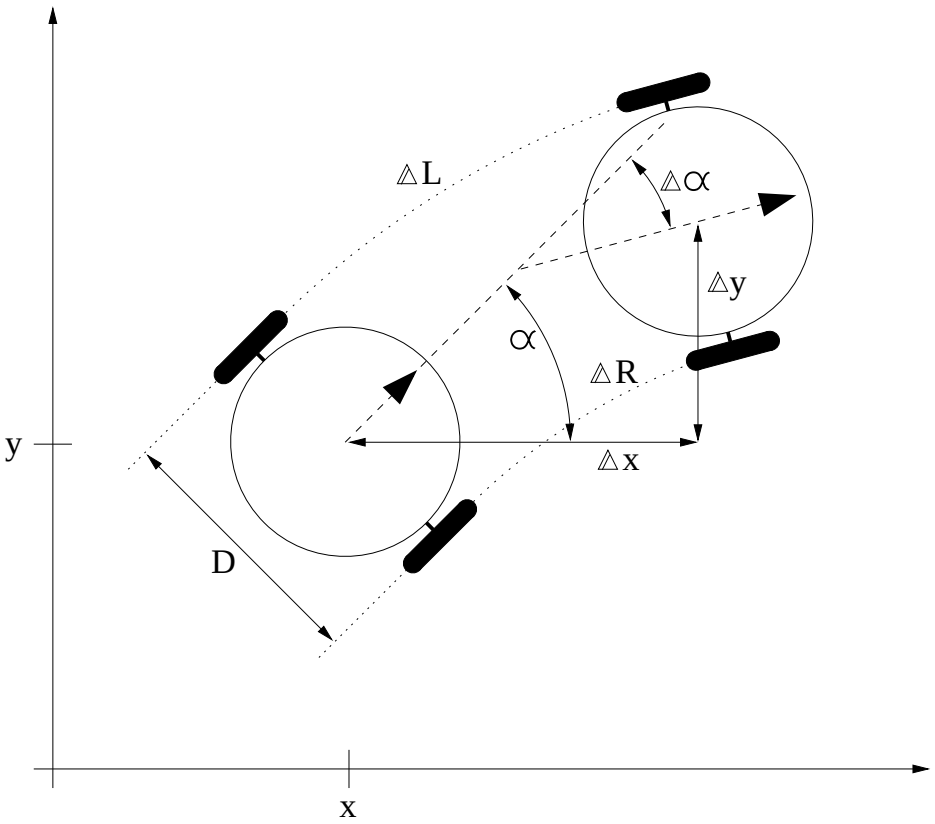


Abbildung 2.2: Odometrie

dinatensystem schematisch dargestellt. Die Position des Roboters ist durch das Tupel $P_{robot} = \{x, y, \alpha\}$ gegeben, die Werte ΔL und ΔR werden direkt von den Radwegsensoren (Umdrehungszähler) abgelesen.

Der Drehwinkel $\Delta\alpha$ ist die Wegdifferenz des rechten und linken Rades geteilt durch den Radabstand D.

$$\Delta\alpha = \frac{\Delta R - \Delta L}{D} \tag{2.1}$$

Die Wegstrecke Δs ist der mittlere Weg

$$\Delta s = \frac{\Delta L + \Delta R}{2} \tag{2.2}$$

Die Positionsveräderung bei Geradeausfahrt ($\Delta L = \Delta R$) ist

$$\Delta x = \Delta R * \cos(\alpha) \tag{2.3}$$

$$\Delta y = \Delta L * \sin(\alpha) \tag{2.4}$$

Die Positionsveränderung bei Kurvenfahrt ($-\Delta L \neq \Delta R$ und $\Delta L \neq 0$) ist

$$\Delta x = \frac{\Delta s}{\Delta\alpha} * \left(\cos\left(\frac{\pi}{2} + \alpha - \Delta\alpha\right) + \cos\left(\alpha - \frac{\pi}{2}\right)\right) \tag{2.5}$$

$$\Delta y = \frac{\Delta s}{\Delta\alpha} * \left(\sin\left(\frac{\pi}{2} + \alpha - \Delta\alpha\right) + \sin\left(\alpha - \frac{\pi}{2}\right)\right) \tag{2.6}$$

2.2 *khepera*-Roboter

2.2.1 Technische Daten

Für die Versuche stand der *khepera*-Roboter, der speziell für wissenschaftliche Arbeiten von der Firma *Applied AI Systems, Inc.* entwickelt wurde, zur Verfügung. Der Name Khepera stammt aus dem Ägyptischen und bedeutet "Käfer". Getreu seinem Namen ist er auch ein Winzling unter den Robotern, so daß er auf dem Tisch eingesetzt werden kann. Er besteht aus einem zylindrischen Körper von ca. 55

mm Durchmesser und 30 mm Höhe. Angetrieben wird der *khepera*-Roboter von zwei Schrittmotoren, die unabhängig voneinander über Untersetzungsgetriebe auf je ein Rad wirken. Vorn und hinten besitzt er einen Teflon-Sporn, auf dem er rutscht. Jedes Rad ist mit einem Zähler gekoppelt, der jeweils nach ca. 0, 08 mm zurückgelegter Wegstrecke um eins erhöht (bei Rückwärtsfahrt erniedrigt) wird. Weiterhin besitzt der Roboter, auf seinen Umfang verteilt, acht Reflexlichtschranken, die wahlweise eine Entfernung zu einem reflektierenden Gegenstand oder die Umgebungslichtintensität messen können. Die Reichweite der Entfernungssensoren beträgt ca. 25 mm. Der Roboter verfügt weiterhin über vier Akkus, mit denen er bis zu einer halben Stunde umherfahren kann.

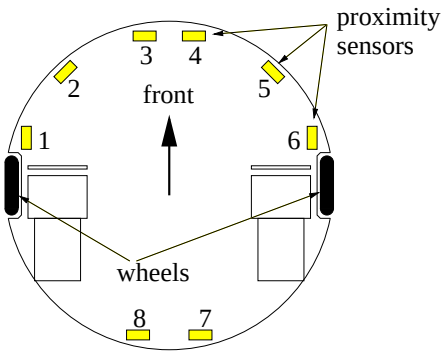


Abbildung 2.3: Prinzipskizze des *khepera*-Roboters

Ein Mikrocontroller vom Typ Motorola 68331, im weiteren Text als Roboter-Controller[1] bezeichnet, übernimmt die Ansteuerung der Komponenten und die Kommunikation mit einem Hostrechner über die serielle Schnittstelle. Über diese Schnittstelle können Daten mit maximal 38400 Baud übertragen werden.

Weiterhin sind für den *khepera*-Roboter eine lineare CCD-Kamera, eine Farbkamera, ein Greifer und ein IO-Modul für eigene Erweiterungen als Zubehör erhältlich.

2.2.2 Betriebsarten des Roboters

2.2.2.1 Kabelfernsteuerung

Bei der Kabelfernsteuerung übernimmt der Hostrechner die gesamte Berechnung und sendet nur einfache Steuerbefehle (z. B. "setze Geschwindigkeit für linkes Rad auf 5, 0") an den Roboter. In jedem Zeitschritt müssen die Sensoren ausgelesen, die Steuerinformation berechnet und an den Roboter zurückgesendet werden. Darüber hinaus müssen auch die (sofern vorhanden) Lernalgorithmen des gewählten Controllers ausgeführt werden. Vorteile dieser Betriebsart sind die Nutzung von Hochsprachen (z. B. C++) bei der Programmierung, die permanente Stromversorgung des Roboters, die hohe Rechenleistung des Hostrechners und die Möglichkeit der permanenten Überwachung des Versuchs und Aufzeichnung der Robotersensoren. Nachteile dieser

[1] Gemeint ist der auf dem Roboter in Hardware realisierte Controller im Gegensatz zu Softwaremodulen, die auf dem Hostrechner laufen und auch als Controller bezeichnet werden.

Betriebsart sind die permanent notwendige und nicht fehlertolerante Kabelverbindung mit dem Hostrechner, die starke Verzögerung des Programmablaufs durch den hohen Kommunikationsaufwand und die eingeschränkte Reichweite des Roboters durch die Kabellänge.

2.2.2.2 Crosskompilierung

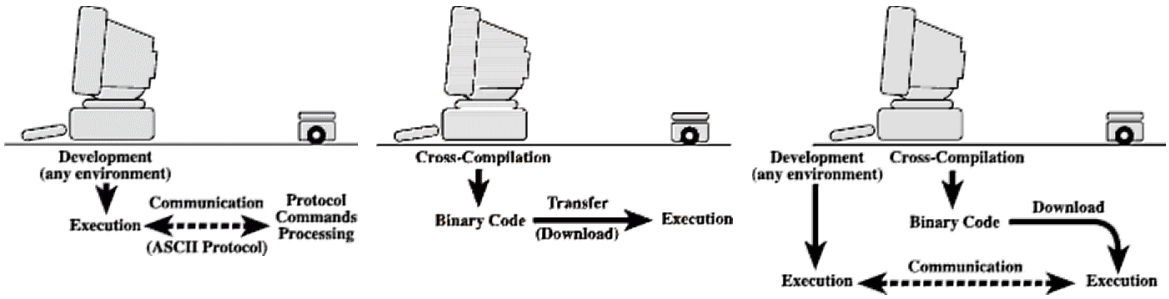
Bei der Crosskompilierung wird auf dem Hostrechner Binärcode für die auf dem Roboter verwendete CPU erzeugt. Vor dem Start des Programms muß dieser Binärcode in den Speicher des Roboters geladen werden. Danach kann das Kabel entfernt werden. Die Vorteile dieser Betriebsart sind die schnellere Ausführung der Steuerung und die erhöhte Bewegungsfreiheit des Roboters. Die Nachteile sind die durch die Batteriekapazität beschränkte Versuchszeit, die hardwarenahe Programmierung in Assembler und der Verzicht auf Datenlogging und Fehlersuchmöglichkeiten. In einigen Fällen werden, bedingt durch das Fehlen einer FPU[2] auf dem Roboter, Berechnungen ungenauer ausgeführt, was sich bei Lernalgorithmen nachteilig auswirken kann.

2.2.2.3 Hybrid-Konzept

Bei der Hybrid-Technik können die Vorteile der beiden oben besprochenen Betriebsarten gemeinsam genutzt werden. Die Steuerung wird crosskompiliert und in den Speicher des Roboters geladen. Die Lernalgorithmen können auf dem Hostrechner mit hoher Geschwindigkeit ausgeführt werden. Von Zeit zu Zeit kommuniziert der Hostrechner mit der CPU des Roboters und führt ein Update der Controllerparameter aus. Die Robotersteuerung wird durch die Kommunikation nicht beeinträchtigt. Die CPU des Roboters kann Sensordaten sammeln und an den Hostrechner übergeben. Die Kabelverbindung zum Roboter muß nicht immer bestehen. Dieser Modus ist für Versuche mit autonomen Robotern am besten geeignet, weil er dem Szenario "Roboter agiert autonom und fährt selbsttätig zu Batterieladestation" am ehesten entspricht. Die Nachteile dieser Betriebsart sind der sehr hohe Implementierungsaufwand und die Notwendigkeit einer gut durchdachten Kommunikationsschnittstelle. In Abbildung 2.4 ist eine Übersicht über die Betriebsarten[3] des *khepera*-Roboters zusammengefaßt.

[2] math. Koprozessor

[3] entnommen aus http://diwww.epfl.ch/lami/robots/K-family/



a) Kabelfernsteuerung *b) Crosskompilierung* *c) Hybrid-Methode*

Abbildung 2.4: Übersicht über die Betriebsarten des *khepera*-Roboters

2.3 Steuerungskonzepte

2.3.1 Konventionelle Kabelfernsteuerung

Im folgenden soll auf einige grundlegende Eigenschaften in Bezug auf die in Punkt 2.2.2.1 erwähnte Steuerung per Kabel eingegangen werden. In den meisten Fällen erfolgt die Steuerung aufgrund vorher eingelesener Sensorwerte, d.h. es handelt sich hierbei um einen *Regelkreis*.

Zur Programmabarbeitung stehen zwei CPUs zur Verfügung, eine meist schnellere auf dem Hostrechner und ein Microcontroller auf dem Roboter. Ein Arbeitsgang kann in folgende Arbeitsschritte unterteilt werden.

1. Sensorwerte einlesen
2. Datentransfer zum Hostrechner (Zeit t_R)
3. Reaktion berechnen (Zeit t_C)
4. Kommandotransfer zum Roboter (Zeit t_T)
5. Aktion ausführen
6. weiter mit 1.

Dieser Zyklus wird als ein Zeitschritt oder Updateintervall bezeichnet. Unter der Annahme eines endlich dauernden Datentransfers zum Roboter repräsentieren die eingelesenen Sensorwerte immer einen Zustand in der Vergangenheit. Werden die Sensorinformationen zur Berechnung einer neuen Steueraktion herangezogen, so erfolgt die

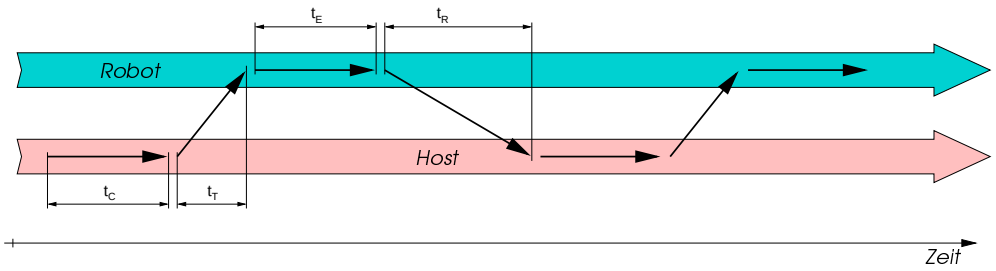


Abbildung 2.5: Programmausführung mit einfachem Controllermodell

eigentliche Aktion, durch den Kommandotransfer zum Roboter verzögert, erst in der Zukunft.

Die verteilte Abarbeitung auf der Host-CPU und dem Robotercontroller ist in der Abbildung 2.5 schematisch skizziert. Die benötigte Zeit für die Einzelschritte 1 und 5 sind in der Zeitangabe t_E zusammengefaßt. Die Summe über die Zeitabschnitte t_E, t_R, t_C und t_T ist die Dauer eines *Zeitschrittes* bzw. die Periodenlänge des Updatezyklus. Die für die Kommunikation benötigte Zeit läßt sich gut aus der verwendeten Baudrate und der übertragenen Datenmenge abschätzen. Die beiden anderen Zeitintervalle können nur durch ausgeklügelte Meßmethoden zur Laufzeit bestimmt werden, die stark vom verwendeten Betriebssystem abhängig sind. Wenn die maximale Baudrate des *khepera*-Roboters von 38400 Baud genutzt wird, sind die erreichbaren Updateintervalle typischerweise 10 bis 20 Millisekunden lang. Die Dauer eines Zeitschrittes wird maßgeblich von der Kommunikationsgeschwindigkeit bestimmt. Durch die relativ hohe Rechenleistung der eingesetzten Prozessoren beträgt die reine Programmabarbeitungszeit (Zeitschritte t_C und t_E) nur ungefähr ein Zehntel der Dauer der Kommunikation (Zeitschritte t_R und t_T). Die Echtzeitfähigkeit wird in diesem Fall weniger durch das Betriebssystem als durch das verwendete Steuerungskonzept eingeschränkt. Ein effektiveres Kommunikationsprotokoll (anstelle des werkseitig eingestellten) kann hier die Feinheit der Steuerung erhöhen.

Weiterhin werden die Bewegungen des Roboters durch Masseträgheit, langsam anlaufende Motoren etc. verzögert, so daß er mitunter erst 3 oder 4 Zeitschritte später auf ein Steuersignal reagiert und eine Rückkopplung über veränderte Sensorwerte erst danach erfolgen kann. Die Reaktion auf ein Ergeignis erfolgt also niemals sofort. Die resultierende Verzögerung ist die Nachrichtenlaufzeit. Auf die Nachrichtenlaufzeit wird in Kapitel 3 noch näher eingegangen.

Kapitel 3

Das Design

3.1 Experimentiersystem

Das Experimentiersystem besteht aus drei Hauptkomponenten, dem Hostrechner, der Software, der Umgebung für den Roboter und dem Roboter selbst. Die Umgebung ist ein kleines, auf dem Tisch aufgebautes Labyrinth aus Styropor. Darin fährt ein kleiner Miniaturroboter. Der Aufbau trägt Modellcharakter. Auf dem Hostrechner läuft eine Software, die den Roboter während der Durchführung eines Versuchs steuert und überwacht. Die Software unterteilt sich in

1. Bibliothek zur Ansteuerung der Funktionsprimitive des Roboters
2. Steuerungs- und Überwachungssoftware
3. Entwicklungswerkzeuge.

Software-Entwicklungswerkzeuge sind für jedes UNIX-Betriebssystem verfügbar. Sie werden in dieser Arbeit nicht näher beschrieben.

3.2 Anforderungen an die Software des Experimentiersystems

1. Visualisierung der Ergebnis- und Sensordaten

Um einen Versuch effizient überwachen zu können, ist es notwendig, daß Variablen des zu erprobenden Algorithmus angezeigt werden. Die versuchsdurchführende Person kann eventuelle Fehler im Algorithmus frühzeitig erkennen und den Versuch stoppen, wenn die Werte nicht im erwarteten Bereich liegen. Die Sensordaten können zu Plausibilitätsbetrachtungen herangezogen werden und vermitteln die interne Sicht des Roboters.

2. Aufzeichnung / Statistiken

   (a) Versuchsdaten

      i. Sensordaten, für Simulationsläufe
      In [NML95] wird gezeigt, daß Roboterevolution sehr wirklichkeitsnah simuliert werden kann, wenn Serien von "echten" Sensordaten als Eingabe benutzt werden. Die Software soll bei einer Roboterfahrt Serien von Sensorwerten aufnehmen können, um damit später Simulationen durchführen zu können.

      ii. von Laufzeitalgorithmen berechnete Daten und Resultate, z.B. Roboterposition

      iii. Erfassung aller Daten pro Zeitschritt
      Um die Auswertung der Logdaten zu erleichtern, müssen sie zeitlich zuzuordnen sein. Möglich ist die Abspeicherung mit Zeitstempel, oder es wird in jedem Zeitschritt ein kompletter Datensatz geschrieben.

   (b) Bildschirmausgaben, Vorführung

   Das Mitloggen der Bildschirmausgaben ist eine zusätzliche Funktion, die für Demonstrationen und Vorführungen interessant ist.

   (c) System-inhärente Parameter zur Plausibilitäts- und Qualitätsabschätzung

      i. Updatefrequenz, Feinheit der Steuerung
      Die Updatefrequenz bzw. die Zeitdauer einer Sensor-Motor-Koordination bestimmt die Feinheit der Steuerung und darüber die maximal nutzbare Geschwindigkeit. Fährt der Roboter so schnell, daß er in einem Zeitschritt mehr Weg zurücklegt als seine Sensorreichweite beträgt, kann der beste Steuerungsalgorithmus nicht mehr funktionieren. Das System muß echtzeitfähig sein, die maximal zulässige Zeittoleranz wird in diesem Fall von der Robotergeschwindigkeit bestimmt. Höhere Geschwindigkeiten des Roboters erfordern kürzere Zeitschritte bei der Steuerung.

ii. Prozessorlast mitloggen

Eine hohe Prozessorlast bedeutet, daß ein Task-Switch eventuell erst später ausgeführt werden kann. Die Antwort der Software auf externe Ereignisse (Interrupt von der seriellen Schnittstelle) kann sich damit verzögern. Ein unter hoher Prozessorlast durchgeführter Versuch ist sicherlich nicht optimal, aber unter bestimmten Umständen läßt sich eine hohe Prozessorlast nicht immer vermeiden. Tritt bei einem Langzeitversuch kurzzeitig eine hohe Prozessorlast auf (z.B. durch einen regelmäßig gestarteten Backup-Prozeß), kann das Mitloggen der Prozessorlast bei der Interpretation der Ergebnisse nützlich sein.

iii. Kommandoverzögerung, Ansprechverhalten und Kontinuität der Steuerung

Wenn auf dem Hostrechner ein Multitasking-Betriebssystem läuft, können Ereignisse, die nichts mit dem eigentlichen Versuch zu tun haben, Einfluß auf die Ergebnisse nehmen. Um diesen Einfluß gering zu halten, kann die Software mit einer höheren Priorität laufen als andere Prozesse. Das ist nicht immer möglich. Deshalb ist es notwendig, wenigstens für die Interpretation der Ergebnisse zusätzliche Anhaltspunkte zu haben. Steigt zum Beispiel durch hohe Prozessorlast die Abarbeitungsdauer eines Arbeitsschrittes (Updateintervall) an, legt der Roboter in dieser Zeit auch einen längeren Weg zurück. Folglich sind die Änderungen der Sensorwerte von einem Zeitschritt zum nächsten auch größer. Das kann dazu führen, daß ein Algorithmus schneller (evtl. zu schnell) oder langsamer konvergiert. Werden aus diesem Versuch initiale Parameter für Verstärkungsfaktoren oder Lernraten abgeleitet, würden sie in diesem Fall falsch gewählt werden. Die günstigste Verfahrensweise wird erreicht, wenn in den zu untersuchenden Algorithmus die aktuelle Updatefrequenz in irgendeiner Art und Weise mit einfließt.

3. Langzeit-Funktionsfähigkeit

Wegen der schon erwähnten Verzögerungen ist es oft notwendig, einen Versuch über einen längeren Zeitraum durchzuführen. Eventuell wird festgestellt, daß eine Variable zu Beginn des Versuchs falsch gewählt wurde. Um den Versuch nicht von vorn anfangen zu müssen, und damit andere Ergebnisse zu verlieren, soll es möglich sein, Parameter während des Versuchs zu ändern. Desweiteren kann es notwendig werden, einen Versuch "einzufrieren", um ihn zu einem späteren

Zeitpunkt fortzusetzen. Die Möglichkeit, einen Versuch auch von einer entfernten Arbeitsstation aus zu überwachen, kann für die Langzeitfunktionsfähigkeit ebenfalls von Bedeutung sein.

4. Automatisierung

   Für immer wiederkehrende Serien von Experimenten (z.B. bei Roboterevolution) wird ein hoher Grad der Automatisierung der Versuchsdurchführung angestrebt. Dauert eine Versuchsreihe über einen längeren Zeitraum an, kann die versuchsdurchführende Person nicht immer anwesend sein. Das Experiementiersystem soll automatisch Versuche überwachen und gegebenenfalls neu starten können.

5. Fehlerdetektion

   Typische, häufig auftretende Fehler sollen erkannt werden, um einerseits zusätzliche Informationen für die Algorithmen zur Verfügung zu haben und um andererseits die Hardware des Roboters vor Beschädigungen durch fehlerhaft arbeitende Algorithmen zu schützen. So kann zum Beispiel die Information "rechter Motor blockiert" für einen Navigationsalgorithmus wertvoll sein. Die Bereitstellung dieser Information erfolgt quasi über einen virtuellen Sensor. Weiterhin kann diese Information zur automatischen Bewertung einzelner Algorithmen herangezogen werden (Algorithmus b hat n-mal die Motoren zum Blockieren gebracht). Für den Fall, daß die Hardware des Roboters keinen Überlastungsschutz besitzt, kann der Versuch auch abgebrochen werden, wenn ein Motor länger als n Sekunden blockiert ist.

6. Wiederholung/Simulation

   Werden komplette Sensorwert-Trajektorien aufgezeichnet (siehe Punkt 2(a)i), ist es auch möglich, Simulationen durchzuführen, indem die aufgezeichnete Trajektorie wieder abgespielt wird. Die Eingabedaten für die Algorithmen kommen dann nicht von den Sensoren, sondern aus der Aufzeichnung. Damit kann ein und derselbe Versuch mit leicht geänderten Parametern in der Simulation mit großem Zeitvorteil wiederholt werden.

7. Gleichzeitiger Test mehrerer Algorithmen / Evaluation mehrerer Agenten

   Aufgrund der geringen Ausführungsgeschwindigkeit bei Echtzeitexperimenten soll die Software in der Lage sein, mehrere Algorithmen/Agenten gleichzeitig zu evaluieren. Algorithmen, die den Roboter nicht selbst steuern, können problemlos parallel berechnet werden. Algorithmen, die auch auf die Aktuatoren

des Roboters zugreifen, müssen nacheinander getestet werden (z.B. jeder Algorithmus steuert den Roboter für n Updateschritte). Die Software benötigt dazu einen Mechanismus, um die einzelnen Algorithmen/Agenten auszutauschen und zu bewerten. Dieser Mechanismus wird auch bei der Durchführung von Evolutionsalgorithmen[1] benötigt.

8. Software- Programmarchitektur

   Die Software soll leicht modifizierbar und erweiterbar sein. Sie sollte weitgehend unabhängig vom verwendeten Betriebssystem sein. Auch eine Portierung für andere Roboterarchitekturen sollte einfach möglich sein.

9. leichte Bedienbarkeit, grafische Oberfläche

   Eine grafische Oberfläche ist notwendig, um Versuchsdaten (z.B. 2D-Karten, neuronale Netze usw.), gut visualisieren zu können. Leichte und übersichtliche Bedienbarkeit wird von jeder Software gefordert.

3.3 Objektorientiertes Controller-Modell

3.3.1 Laufzeitmodul

Das Laufzeitmodul ist der Teil des Codes, der für einen bestimmten Algorithmus pro Zeitschritt abgearbeitet werden soll. Da mehrere Algorithmen zur selben Zeit berechnet und außerdem noch eine Ausgabe der Werte auf dem Bildschirm erfolgt, muß dieser Teil so schnell wie möglich abgearbeitet werden. Er darf nicht blockieren oder warten, und er sollte eine festgelegte Abarbeitungsdauer nicht überschreiten. Ein Laufzeitmodul wird typischerweise innerhalb der Hauptschleife eines Programms aufgerufen und beinhaltet den Code für einen speziellen Controller-Agent.

3.3.2 Controller-Agent

Der Controlleragent ist ein Softwareobjekt, das einen konkreten Algorithmus implementiert. Es liest Sensorwerte und steuert den Roboter. Mehrere Controller-Agenten können parallel[2] ausgeführt werden. Der Prototyp für einen Controller-Agenten wurde

[1] Algorithmen, die Evolutionsprozesse im Computer nachbilden
[2] nacheinander in einem Zeitschritt

hinsichtlich späterer Erweiterungen auf andere Steuermodi sehr modular und offen gestaltet. Alle Methoden eines Controller-Agenten werden vom Metacontroller initiiert. Jeder Controller-Agent besitzt Basisfunktionalität in folgenden Methoden:

1. Start/Stopp
2. Pause
3. Berechnung
4. Ausführung
5. Anzeige
6. Logging
7. Parameterauswahl.

Erweiterungen und Spezialisierung der Controller-Objekte werden durch Ableitung von Objekten und Überlagerung der virtuellen Methoden erreicht. Das gesamte Design ist darauf ausgerichtet, daß ein Programmierer einfach und mit wenig Aufwand neue Ideen implementieren kann, ohne wiederkehrende Aufgaben doppelt implementieren zu müssen.

Die Trennung von Berechnung und Ausführung erlaubt dem Metacontroller, die Berechnungsmethode zu einem Zeitpunkt auszuführen, an dem keine anderen zeitkritischen Routinen ausgeführt werden müssen (z.B. in der Zeit, in der auf Daten vom Roboter gewartet wird). Weiterhin ist es durch Überlagerung der Ausführungsmethode möglich, das unter Punkt 2.2.2.3 beschriebene Hybrid-Konzept zur Steuerung des Roboters zu verwenden. Die Ausführungsmethode wird einmal pro Zeitschritt angesprochen, wenn der Controlleragent die Steuerung des Roboters übertragen bekommen hat. Jeder Controller-Agent besitzt eigene Methoden zur Nutzer-Interaktion. Algorithmenspezifische Parameter lassen sich damit auch während der Laufzeit ändern. Ein Bitfeld spezifiziert die Eigenschaften bezüglich der möglichen Nutzerinteraktionen. Alle Controller-Agenten beziehen und senden ihre Daten nicht direkt an den Roboter, sondern an das in Abschnitt 3.5 beschriebene Kommunikationsmodul.

3.4 Meta-Controller

3.4.1 Aufgaben

Der Metacontroller dient dem Management aller Controlleragenten und spielt bei der

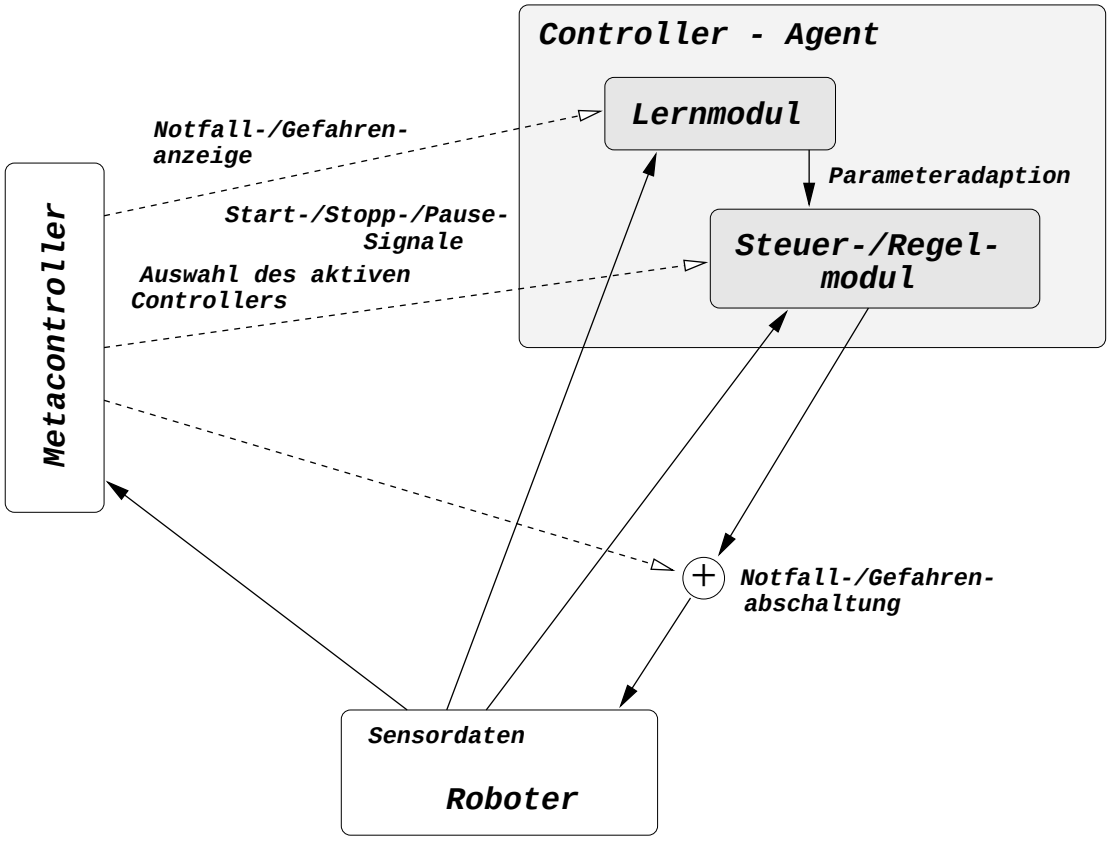


Abbildung 3.1: Kommunikation zwischen Roboter, Controlleragent und Metacontroller

Automatisierung der Versuche eine große Rolle. Die Controlleragenten werden in einer verketteten Liste verwaltet. Die Methoden der Controlleragenten werden nur über den Metacontroller aktiviert. Der Metacontroller übernimmt die Kollisionsdetektion und schaltet den Versuch ab, wenn eine Beschädigung der Roboterhardware angenommen wird. Weiterhin mißt er die Ausführungszeiten der Controlleragenten und die Antwortzeiten des Roboters und stellt diese Information allen Controlleragenten zur Verfügung. In Abbildung 3.1 ist der Informationsaustausch zwischen Metacontroller, Controlleragent und Roboter dargestellt.

3.4.2 Kollisionsdetektion

Die Erkennung einer Kollision oder Blockade der Räder ist sehr wichtig. Der Metacontroller stellt diese Information den lernenden Agenten zur Verfügung und unterbindet weitere Fehlfunktionen. Ein einfacher Algorithmus zur Erkennug blockierter Räder wird hier vorgestellt.

- Die vom jeweiligen steuernden Agenten gewünschte Radgeschwindigkeit v_s wird über einen längeren Zeitraum (ca. 50 Zeitschritte) gemittelt,
- die zurückgelesene tatsächliche Radgeschwindigkeit v_r wird ebenfalls gemittelt,
- falls die Differenz der gewünschten und der tatsächlichen Radgeschwindigkeit größer ist als zwei Drittel der gewünschten Geschwindigkeit, liegt eine Blockierung des jeweiligen Rades vor.

$$Bool_{crash} = |\overline{\overline{v_s} - \overline{v_r}}| > \frac{2}{3} |\overline{v_s}| \tag{3.1}$$

In Abbildung 3.2 ist der Verlauf der Parameter v_s und v_r in einer Crash-Situation geplottet. Zuerst bewegte sich der Roboter nahezu geradeaus mit kleineren Schwankungen der Radgeschwindigkeit. Dann folgt ein kurzes Abbremsen, eine etwas größere Beschleunigung bis auf ca. $6,5\frac{mm}{s}$ und wieder ein Abbremsen. Im Zeitschritt 310 blockiert das Rad. Das Mittel über die zurückgelesene Radgeschwindigkeit fällt auf Null, und die Differenz der Radgeschwindigkeiten steigt langsam über den Schwellwert. Nach dem Zeitschritt 420 ist die Blockade des Rades zuverlässig erkannt.

3.4.3 Zusammenwirken der Controller-Agenten

Der Metacontroller schaltet immer nur einen Controlleragenten zur Steuerung des Roboters frei. Die Berechnungsmethoden der Controlleragenten werden jedoch immer aktiviert. Die Auswahl des aktiven Controlleragenten kann durch Nutzereingabe, nach Zeitsteuerung oder durch Heuristiken (z.B. welcher Controller verursacht die wenigsten Crashs) geschehen. Komplexere Unterfunktionen des Metacontrollers (z.B. das Auflösen einer Crashsituation) können ebenfalls durch Controlleragenten implementiert werden, die unabhängig von den zu testenden Controlleragenten immer vorhanden bleiben. Das Schema in Abbildung 3.3 zeigt die Hierarchie der einzelnen Controllermodule.

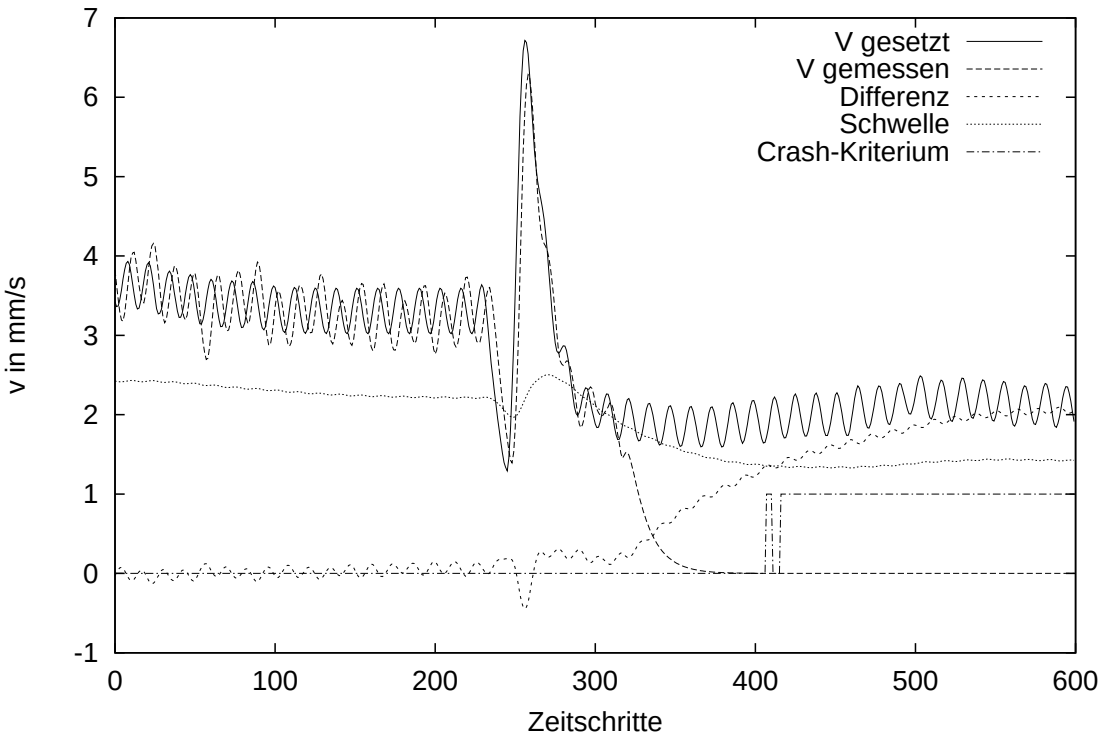


Abbildung 3.2: Radgeschwindigkeiten in Crash-Situation

Wird der Metacontroller mit einer normierten Bewertungsfunktion für alle verwalteten Agenten ausgestattet, können sehr komplexe Verhaltensweisen modelliert werden. Die einzelnen Controlleragenten können sich dann auf Grundfunktionen spezialisieren und werden immer dann eingeschaltet, wenn sie sich gerade am besten bewähren. Die Auswahl des aktiven Controlleragenten kann z.B. über die Fisher-Eigen-Dynamik [EEF90] erfolgen. Damit können lernende Controller automatisch generiert werden.

Noch komplexeres Verhalten kann erreicht werden, wenn die Struktur geschachtelt wird, also jeder Controlleragent wiederum einen Metacontroller und mehrere Unteragenten besitzt.

3.5 Kommunikationsmodul

Wegen der Möglichkeit, mehrere Algorithmen parallel auszuführen, ist eine gemeinsame Kommunikationsschnittstelle zum Roboter notwendig. Um die Programmausführung nicht durch zwei gleiche Abfragen pro Zeitschritt zusätzlich auszubremsen, wurde zwischen Steuerung und Roboter noch eine Schicht etabliert, die dafür sorgt,

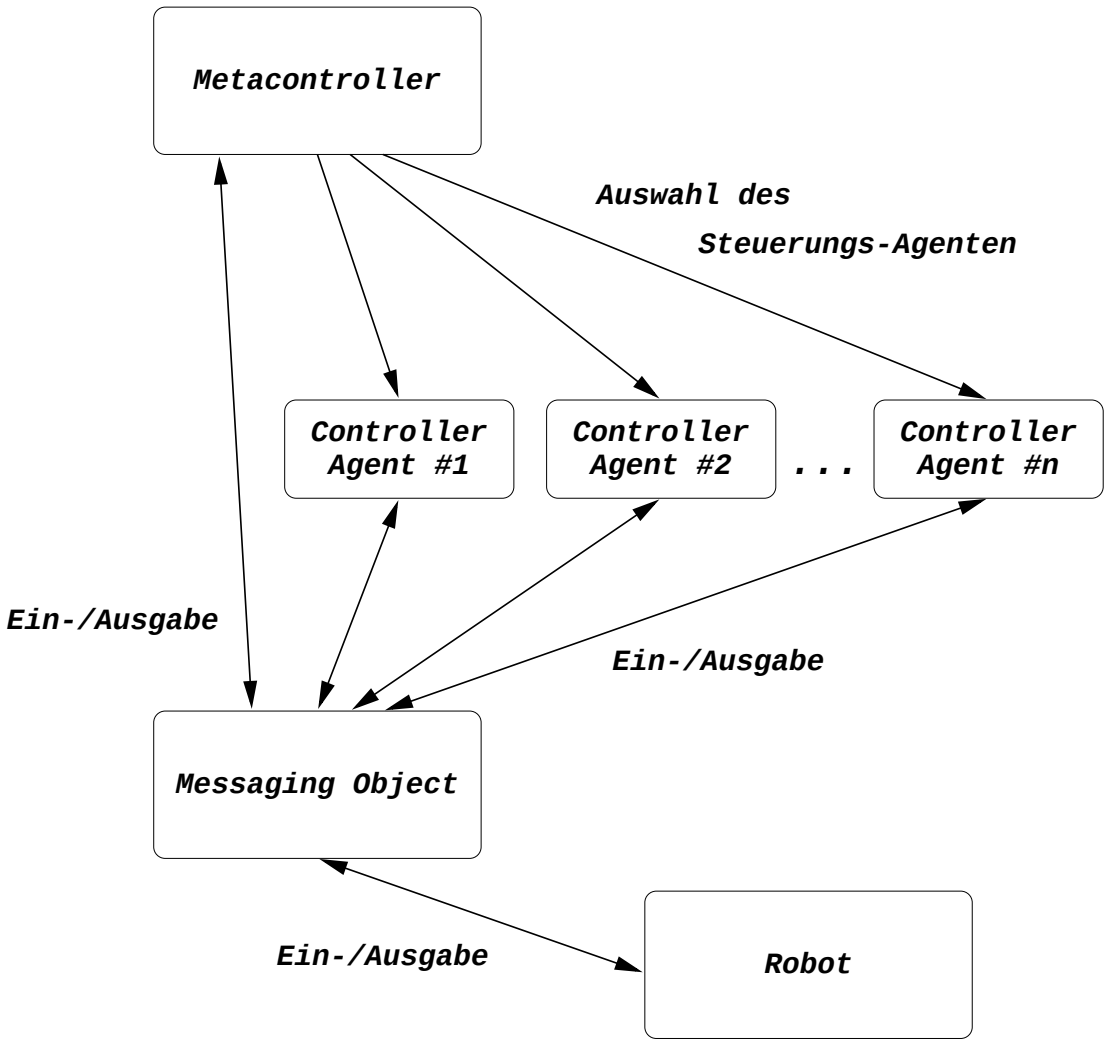


Abbildung 3.3: Agentenaustausch

daß doppelte Anfragen eliminiert und ausgelesene Werte gepuffert werden (*messaging object* in Abbildung 3.3). Wenn ein Controller-Agent einen bestimmten Sensorwert lesen will, meldet er das durch Setzen des jeweiligen Bits im Nachrichten-Objekt an. Einmal pro Zeitschritt wird mit Hilfe der gesetzten Bits eine Anfrage an den Roboter generiert und alle Bits gelöscht. Wenn Daten an den Roboter übertragen werden müssen, werden sie erst in einem Puffer im Nachrichten-Objekt zwischengepeichert. Vor dem Absenden werden doppelte Einträge aus dem Puffer entfernt. Wird zum Beispiel "Setze Radgeschwindigkeit" zweimal pro Zeitschritt im Sende-Puffer gefunden, so wird nur der letzte Eintrag berücksichtigt. Empfangene Daten werden in einem Lese-Puffer zwischengespeichert, auf den alle anderen Objekte des Programms zugreifen dürfen.

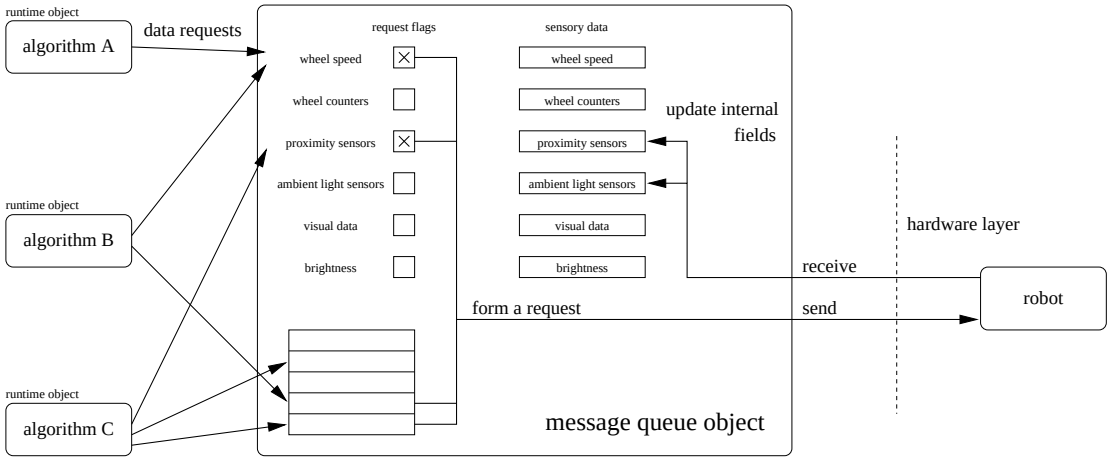


Abbildung 3.4: Nachrichten-Objekt

Die Vorteile dieser Kommunikationsschnittstelle sind:

- einheitliche Software-Schnittstelle für alle Controlleragenten, Lesecache für Mehrfachleseoperationen
- klare Diskretisierung des Zeitraumes, einheitliche Abfrage aller Sensoren (die Zeitspanne von der Abfrage des ersten bis zum letzten Sensor ist möglichst kurz)
- Synchronisation der Ein-/Ausgabeoperationen mit den Zeitschritten
- Freistellung von Leerlaufzeiten durch nicht blockierende IO-Operationen. Es ist mehr Zeit für Berechnungen verfügbar, da nicht auf Input gewartet werden muß.
- Unterstützung der Zweipfadabarbeitung (im nächsten Abschnitt erläutert)
- Verringerung der Redundanz in den Anfragen, Verringerung des Kommunikationsaufwandes, Vermeidung von "Überlastung" der Roboterhardware durch sehr kurz aufeinanderfolgende Anfragen.
- optimale Ausschöpfung der für die Kommunikation zur Verfügung stehenden Bandbreite

Ein Nachteil ist die Verlängerung der Nachrichtenlaufzeit. Auf diese wird im folgenden Abschnitt näher eingegangen.

3.6 Antwortzeitverhalten

3.6.1 Nachrichtenlaufzeiten

Die Ein- und Ausgabe vom steuernden Programm zum Roboter-Controller unterliegt einer signifikanten Verzögerung, die stark vom Betriebssystem des Hostrechners, seiner Architektur, der Bandbreite der Kommunikationsleitung zum Roboter-Controller und der Ausführungsgeschwindigkeit des Roboter-Controllers abhängt. Weiterhin liefern die Masseträgheit und die Antwortzeiten des auf dem Roboter implementierten Regelkreises einen unbekannten Beitrag zum Antwortzeitverhalten des Roboters. Die Nachrichtenlaufzeit kann als roboterspezifische Eigenschaft angesehen werden.

In einem Algorithmus kann es notwendig werden, die Ausgabewerte des Controllers mit zurückgelesenen Sensordaten zu korrelieren. Zwischen den Ausgabedaten und den zurückgelesenen Sensorwerten können aber, bedingt durch die Nachrichtenlaufzeiten, mehrere Zeitschritte liegen. Für Lern- oder Steuerungsalgorithmen kann dieser Wert von großer Bedeutung sein. Deshalb soll der Wert dieser Verzögerung laufend errechnet werden, um für die korrekte Korrelation von Sensordaten und Steuerungsdaten zur Verfügung zu stehen.

3.6.2 Ermittlung der Nachrichtenlaufzeit

Angenommen, die Steuerung gibt die Werte für Geschwindigkeit v und Kurvenradius[3] ω aus. Die Einzelradgeschwindigkeiten sollen wie folgt berechnet werden[4]

$$v_L = (1 + \omega) * v \tag{3.2}$$

$$v_R = (1 - \omega) * v \tag{3.3}$$

Die Krümmung ω soll den Wert $0,0$ bei Geradeausfahrt und den Wert $1,0$ bei einer Kurve mit Radius gleich dem halben Radstand[5] haben. Negative Werte bedeuten eine Kurve entgegen dem Uhrzeigersinn, ein Wert $\pm\infty$ bedeutet Drehen um den eigenen Mittelpunkt. Die dem Roboter-Controller übergebenen Radgeschwindigkeiten werden modifiziert, daß der Kurvenradius, den der Roboter im aktuellen Moment fahren soll, sich aus dem Radius der Vorgabe und einer überlagerten Sinusschwingung

[3] Der Wert ω wird als Krümmung des eigentlichen Radius der Kurve definiert. $\omega = \frac{1}{r}$

[4] Zwischen (v_L, v_R) und (v, ω) kann o.B.d.A. transformiert werden.

[5] rechtes Rad fährt vorwärts, linkes Rad steht

zusammensetzt.

$$\omega_{mod}(i) = \varepsilon * \sin\left(\frac{i * 2\pi}{T}\right) \tag{3.4}$$

$$\omega = \omega_{vorgabe} + \omega_{mod} \tag{3.5}$$

Der Wert i ist der Index des aktuellen Zeitschrittes, der Wert T bestimmt die Frequenz, der Wert ε die Stärke der überlagerten Sinusschwingung. Die gewünschten sowie die zurückgelesenen Radgeschwindigkeiten werden fortlaufend aufgezeichnet. Die überlagerte Sinusfunktion kann in dem Signal der zurückgelesenen Radgeschwindigkeiten mit einer durch die Verzögerung bedingten Phasenverschiebung wiedererkannt werden. Mit Hilfe der Kreuzkorrelation von gesendetem ω_{Mod} und gelesenem Signal ω_{Read} kann die Phasenverschiebung berechnet und damit die Verzögerung der Ein-/Ausgabeoperationen zum Roboter-Controller ermittelt werden.

$$k\,(x) = \sum_{t=0}^{n} \omega_{mod}(t) * \omega_{read}(t + x) \tag{3.6}$$

Der Wert der Phasenverschiebung ist der Wert x, beim dem die Funktion $k(x)$ ihr Maximum hat. Der Wert n ist die Anzahl der in die Betrachtung einbezogenen Zeitschritte.

Die Überlagerung der Steuerungsfunktion mit einer zweiten Funktion kann sich störend auf die Steuerungsalgorithmen auswirken und sollte daher relativ klein sein (max. 3-8%, $\varepsilon = 0,03$). Die Periode T der überlagerten Sinusschwingung sollte ein Vielfaches von der zu erwartenden Verzögerung sein, um Mehrdeutigkeiten auszuschließen. Ebenfalls muß die Periode groß genug gewählt werden, damit die Hardware auch auf diese Sinusschwingung reagieren und ihr folgen kann.

In Abbildung 3.5 ist die angeforderte (durchgezogene Linie) und die zurückgelesene Radgeschwindigkeit (gestrichelte Linie) für ein Rad aufgezeichnet worden. Die überlagerte Sinusschwingung und die Phasenverschiebung des zurückgelesenen Signals ist deutlich zu erkennen. Aufgezeichnet wurde über eine Strecke von ca. 50 cm, die größeren Ausschläge gegen Ende resultieren aus einer Kurve, die der Roboter gefahren ist. Die mittlere Geschwindikgeit beträgt ca. 3 mm/s. Negative Radgeschwindigkeiten bedeuten ein Rückwärtsfahren des Rades. Durch den Betrieb mit langsamer Geschwindigkeit und der daraus folgenden Trunkierung der Geschwindigkeitswerte mußte der Wert für ε aus Gleichung 3.4 auf $0,4$ erhöht werden. Während des Versuchs war eine deutliche Pendelbewegung[6] des Roboters sichtbar.

[6]In manchen Fällen kann es aber auch wünschenswert sein, die Bahn des Roboters mit einer

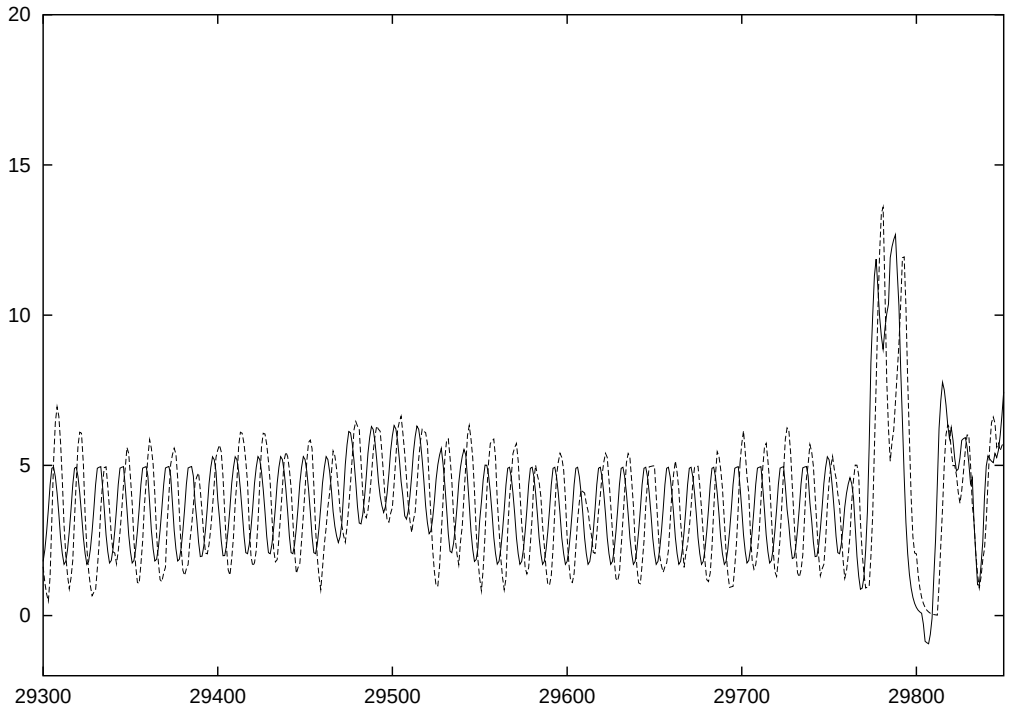


Abbildung 3.5: mit Sinusfunktion modulierte Radgeschwindigkeit

3.6.3 Kompensation der Nachrichtenlaufzeiten

Es wird über die ermittelten Nachrichtenlaufzeiten ein Mittelwert gebildet. Ein typischer Wert beim *khepera*-Roboter sind 2-3 Zeitschritte. Alle Sensordaten werden im Ein-/Ausgabemodul in einem eigenen Ringpuffer zwischengespeichert. Jeder Algorithmus kann damit auch auf die Sensordaten des Zeitschrittes *t-1, t-2 .. t-n* zugreifen. Der Wert n muß kleiner sein als die Größe des Ringpuffers. Die Wahl, welche Sensordaten aus welchem Zeitschritt zur Berechnung im Algorithmus herangezogen werden, liegt beim Designer des jeweiligen Algorithmus, üblicherweise werden Steuersignale des Zeitschrittes *t-Nachrichtenlaufzeit* mit aktuellen Sensordaten aus Zeitschritt t korreliert.

3.7 Erweitertes Ein-/Ausgabemodell

Unter der Annahme, daß der Datentransport auf der Hardware-Ebene unabhängig von der CPU und bidirektional (vollduplex) erfolgt, kann das in 2.3.1 genannte Schema

Sinusfunktion zu modulieren. Im Kapitel 5 ist ein Beispiel dafür gegeben, bei dem diese Modulation gezielt im Lernalgorithmus ausgenutzt wird.

erweitert werden, so daß doppelt soviel Daten übertragen werden können und keine der CPUs längere Pausen zur Synchronisation einlegen muß. Der nächste Arbeitsschritt wird dabei schon um eine halbe Schrittweite eher angefangen. Beide CPUs arbeiten

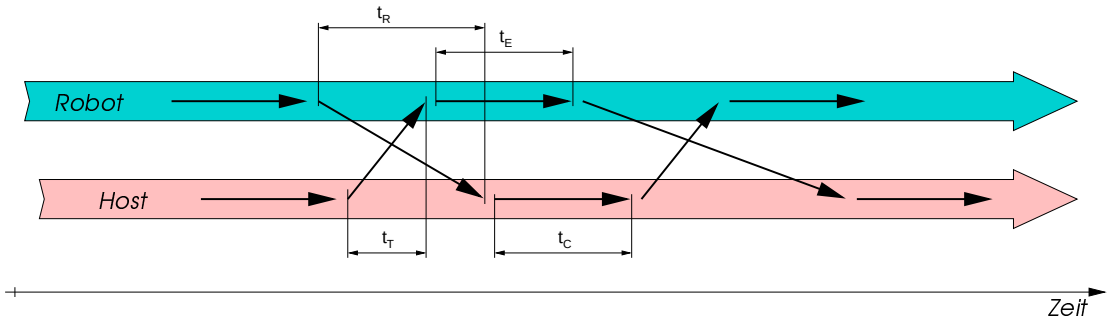


Abbildung 3.6: Vollduplex-Steuerung

um eine halbe Periode zeitversetzt quasi parallel.

Vorteile

- Verbesserung der Feinheit (Granularität) der Steuerung
- Verringerung der Wartezeiten beider CPUs, parallele Verarbeitung

Nachteile Die CPU des Hostrechners erhält pro Zeitschritt nicht mehr die Daten, die als eine unmittelbare Reaktion auf Steuerbefehle angesehen werden können, sondern eventuell auch Daten aus vorangegangenen Zeitschritten. Die Nachrichtenlaufzeit der Daten muß in diesem Fall mit berücksichtigt werden.

3.8 Spezielle Controllermodule

3.8.1 Lichtkompaß

Beim Lichtkompaß wird eine feststehende, weit entfernte Lichtquelle zur Orientierung des Roboters genutzt. Neben der Odometrie existiert damit eine weitere Informationsquelle für die momentane Ausrichtung des Roboters im Koordinatensystem. Die Werte der Umgebungslichtsensoren werden durch eine Kreuzkorrelation mit einer Gauss-Kurve auf einen Winkel abgebildet. Aus diesem Winkel "sieht" der Roboter eine Lichtquelle.

$$g\left(x,s\right)=\exp\left(-\left(\left(x\div 2\pi\right)-s\right)^{2}\right) \tag{3.7}$$

$$k(\alpha) = \sum_{i=1}^{n} x_i * g(Angle(i), \alpha) \tag{3.8}$$

Der Operator $\div$ ist die Modulo-Division. Der Wert x_i ist der Wert des i-ten Lichtsensors, $Angle(i)$ bezeichnet die Winkelausrichtung, in der der Sensor i am Roboter montiert ist. Der Wert α, bei dem die Funktion $k(\alpha)$ ihr Maximum hat, ist der Winkel, unter dem die Lichtquelle "gesehen" wird. Durch die geringe Anzahl der Umgebungslichtsensoren ist die Winkelauflösung relativ grob ($\approx \frac{\pi}{4}$ bei 8 Sensoren). Der Fehler ist aber nicht von der Anzahl der Drehungen oder der Länge des abgelaufenen Weges des Roboters abhängig, sodaß der Lichtkompaß gut zur Korrektur der Positionsinformation (z.B. mit Kalman-Filterung [May90]) eingesetzt werden kann.

3.8.2 Neuronalcontroller

Mit dem Neuronalcontroller kann die Steuerung des Roboters durch ein neuronales Netz erfolgen. Das Netz kann ein Feed-Forward- oder ein rekurrentes Netz sein, welches mit der Software SNNS erzeugt und trainiert wurde.

3.8.3 Pfadsimulation

Bei der Pfadsimulation werden die in einem vorherigen Lauf aufgezeichneten Positions- und Sensorinformationen schnell abgespielt, ohne das ein Roboter angeschlossen ist. Alle anderen Controllermodule erhalten diese Informationen, als kämen sie von einem realen Roboter.

3.8.4 Weitere Module

- adaptive lernende Differentialgleichungen als Orientierungshilfsmittel
- Lernverfahren für autonome Agenten nach dem Prinzip der *homeokinesis*.

Diese beiden Module werden ausführlich in den folgenden zwei Kapiteln beschrieben.

Kapitel 4

Anwendung 1 Differentialgleichungen im Wettbewerb

4.1 Einleitung

Positionsbestimmung und Navigation sind essentielle Grundaufgaben im Forschungsgebiet "autonome Roboter". Die Koppelnavigation (Odometrie) ist ein bekanntes und mit geringem Hardwareaufwand realisierbares Verfahren zur lokalen Positionsbestimmung. Eine indoor-Umgebung, eines der häufigsten Einsatzgebiete von Robotern, ist meist durch orthogonale Strukturen geprägt. Der im folgenden Kapitel beschriebene Algorithmus macht sich diese Eigenschaften zunutze, um einen Korrekturbeitrag für die fehlerbehaftete Koppelnavigation zu geben. Es werden mit einem Echtzeit-Lernverfahren Vorzugsrichtungen (Kanten) extrahiert, die u.a. auch zur Generierung topologischer Karten benutzt werden können.

4.2 Der Algorithmus

4.2.1 Differentialgleichungen beschreiben Kurvenformen

Bei Betrachtung des zurückgelegten Weges eines Roboters ist festzustellen, daß sich die Bahnkurve aus Elementen einfacher Kurvenformen zusammensetzen läßt. Die einfachste von ihnen ist die Gerade. Mit dem Anstieg der Geraden ist zugleich eine

ganz charakteristische Eigenschaft dieser Spezialform einer "Kurve" in nur einem einzigen Wert repräsentiert. Dazu werden noch die Koordinaten eines einzigen Punktes (Startpunkt) und die Länge des Wegstückes benötigt, um diesen Teil der Bahnkurve eindeutig festzulegen.

Wenn dieser Ansatz auf das Prinzip "Differentialgleichungen beschreiben Kurvenformen" verallgemeinert wird, können komplizierte Kurvenformen dargestellt und in den meist beschränkten Speichermedien der Roboter-CPU effektiv abgespeichert werden. So kann zum Beispiel während der Verfolgung einer geraden Wand die Lage bzw. die Ausrichtung dieser Wand im Koordinatensystem oder bei der Umrundung eines Hindernisses sein Durchmesser festgestellt werden. Die Odometriedaten des Roboters werden dabei gleichzeitig gemittelt, um eventuelle Schlängelbewegungen des Wandfolgealgorithmus auszugleichen. Der aus den Odometriedaten resultierende Fehler kann klein angenommen werden, da nur die relative Positionsveränderung betrachtet wird.

4.2.2 Wettbewerb zwischen Differentialgleichungen

Die Basis des Algorithmus bildet ein Differentialgleichungsobjekt $\Theta = \{\overrightarrow{v}, p, \Phi, E\}$. Es enthält einen Vektor $\overrightarrow{v}$ mit allen Ableitungen, zum Beispiel $v_1 = \frac{\partial x}{\partial s}$, $v_2 = \frac{\partial y}{\partial s}$, die Wahrscheinlichkeit p, daß genau dieses Objekt beim nächsten Update berücksichtigt wird, und die aktuelle Fitness Φ bzw. den momentanen Fehler E. Mehrere Differentialgleichungsobjekte Θ_i, eine Abstandsfunktion ϑ und die durchschnittliche Fitness φ bilden nun einen Pool $\Omega = \{\Theta_i, \vartheta, \varphi\}$, $i \epsilon N$, aus dem in jedem Schritt immer ein Objekt herausgegriffen und einem Update unterzogen wird. Das Objekt mit der höchsten Wahrscheinlichkeit gewinnt den Update seiner Parameter. Die Wahrscheinlichkeiten der einzelnen Objekte werden nach ihren Fitnesswerten bzw. Fehlern eingestellt. Dazu wird mit der geeignet gewählten Abstandsfunktion ϑ festgestellt, wie gut eine Differentialgleichung auf das gerade zurückgelegte Wegstück paßt und der zugehörige Fehler ermittelt. Wenn mehrere Differentialgleichungsobjekte annähernd gleich gut repräsentiert sind, soll kein Update vorgenommen werden. Nach ausreichend vielen Updateschritten sind die Koeffizienten konvergiert und die Differentialgleichungen widerspiegeln die geometrischen Eigenschaften der abgelaufenen Wegstücke.

4.3 Formalisierung am Beispiel

4.3.1 Differentialgleichung für Geraden

Die Differentialgleichung in der Form $\frac{dy}{dx} = m$ ist für unsere Zwecke ungünstig, da $m = \pm\infty$ werden kann, wenn die Gerade parallel zur y-Achse des Koordinatensystems verläuft. Deshalb wird von der gebräuchlichen Form $y = m(x - x_0)$ zur Parameterdarstellung der Gerade übergegangen. Die Variable s ist in diesem Fall die Länge des zurückgelegten Weges auf der Geraden.

$$y = sa_y + y_0 \tag{4.1}$$

$$x = sa_x + x_0 \tag{4.2}$$

Nach Differentation erhalten wir

$$\frac{\Delta y}{\Delta s} = a_y \tag{4.3}$$

$$\frac{\Delta x}{\Delta s} = a_x \tag{4.4}$$

Diese Gleichungen sind nun genau dann erfüllt, wenn der Roboter sich auf einer Geraden bewegt, die durch die Parameter a_x und a_y beschrieben wird. Die Werte Δx, Δy und Δs können direkt aus den Odometriedaten des Roboters entnommen werden. Da es sich um relative Koordinaten handelt, ist der zu erwartende Fehler auch entsprechend klein. Um die Parameter a_x und a_y in kleinen Schritten auf einen gewünschten Wert hinzuführen, benötigt man eine Fehlerfunktion als Maß der Abweichung des aktuell zurückgelegten differentiellen Wegstücks $(\Delta x, \Delta y)$ und den intern angenommenen Werten (a_x, a_y).

4.3.1.1 Fehlerfunktion und Fitness

Mit den Beziehungen aus 4.3 und 4.4 läßt sich die folgende Fehlerfunktion formulieren.

$$E_i = \left(\frac{\Delta x}{\Delta s} - a_{x,i}\right)^2 + \left(\frac{\Delta y}{\Delta s} - a_{y,i}\right)^2 \tag{4.5}$$

Der Index i bezieht sich auf das i-te Differentialgleichungsobjekt. Die Fitness dieses Objektes ergibt sich einfach aus

$$\Phi_i = -\frac{E_i}{4} \tag{4.6}$$

Die mittlere Fitness über alle betrachteten Differentialgleichungsobjekte beträgt

$$\overline{\Phi} = \sum_{i=1}^{n} p_i \Phi_i \tag{4.7}$$

4.3.1.2 Fisher-Eigen-Dynamik

Die Fisher-Eigen-Dynamik [EEF90] beschreibt die Selektionsdynamik in einem Pool von Individuen unterschiedlicher Fitness ϕ_i und Wahrscheinlichkeit p_i. Dabei werden die Wahrscheinlichkeiten p_i so verändert, daß nach hinreichend langer Zeit das Individuum i^* mit der höchsten Fitness selektiert wird, d.h.

$$p_i = \left\{ \begin{array}{ll} 1, & falls\, i = i^* \\ 0, & sonst \end{array} \right\}$$

In jedem Zeitschritt wird ein Update für alle Wahrscheinlichkeiten der Differentialgleichungsobjekte im Pool durchgeführt (Fitness Φ_i nach Gleichung 4.6, n ist Anzahl der Differentialgleichungen).

$$\Delta p_i = \eta * \left(\Phi_i - \overline{\Phi} \right) * p_i \tag{4.8}$$

Die Wahrscheinlichkeiten werden normiert. Es gilt $0 \leq p_i \leq 1$ und $\sum_{i=1}^{n} p_i = 1.0$. Die Geschwindigkeit, mit der die Wahrscheinlichkeiten umschlagen können, kann mit dem Parameter η eingestellt werden ($\eta \epsilon [0.5 \,..\, 5.0]$).

4.3.1.3 Information und Update

Gleichzeitig mit dem Update der Wahrscheinlichkeiten p_i erfolgt ein Update der Koeffizienten der Differentialgleichungsobjekte zur Minimierung des Fehlers E_i. Die Größe des Updateschrittes für das Objekt i wird proportional zur Wahrscheinlichkeit p_i gewählt.

$$\Delta a_{j,i} = \varepsilon * p_i * \left(\frac{\Delta x}{\Delta s} - a_j \right) \quad j = \{x, y\} \tag{4.9}$$

Bessere Ergebnisse werden erzielt, wenn die Wahrscheinlichkeit p_i durch eine monoton wachsende Funktion der Gestalt

$$\sigma(p_i) = 1 - \frac{1}{1 + e^{(p - p_0) * \chi}} \tag{4.10}$$

ersetzt wird, wobei $p_0 \epsilon [0.4 .. 0.7]$ und $1 \leq \chi \leq 25$ gewählt wird.

Wird der Algorithmus einige Zeit aktiviert, während der Roboter an einer längeren geraden Wand entlang läuft, werden die Parameter $a_{x,i}$ und $a_{y,i}$ sukzessive nachgeführt und spiegeln schließlich den Verlauf der Geraden wider. Der Vektor $\begin{pmatrix} a_{x,i} \\ a_{y,i} \end{pmatrix} = \overrightarrow{a_i}$ aus den gelernten Koeffizienten ist dann ein Einheitsvektor entlang dieser Geraden.

Die Information nach SHANNON [SW76] dient dazu, Updates zu unterdrücken, falls mehrere Differentialgleichungen ein bestimmtes Wegstück gleich gut repräsentieren.

$$I = -\frac{1}{n} \sum_{i=1}^{n} p_i log(p_i) \tag{4.11}$$

Ein Update der Koeffizienten wird grundsätzlich nur dann durchgeführt, wenn I kleiner als ein vorgegebener Schwellwert ist. Günstige Werte für diese Schwelle liegen bei $\approx \frac{2}{3}\frac{1}{n} \log(n)$. In Abbildung 4.1 ist der Verlauf der Information über der Zeit bei einer

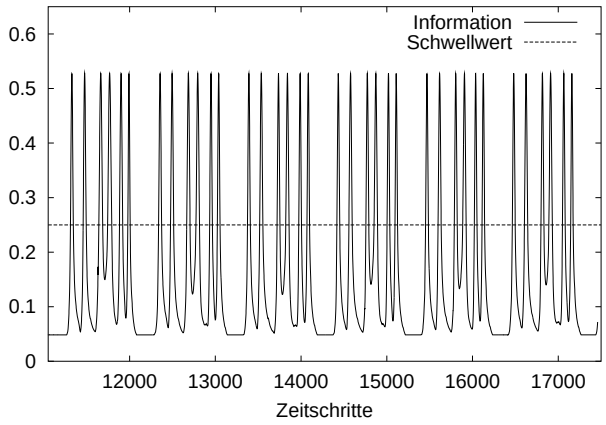


Abbildung 4.1: Verlauf der Information

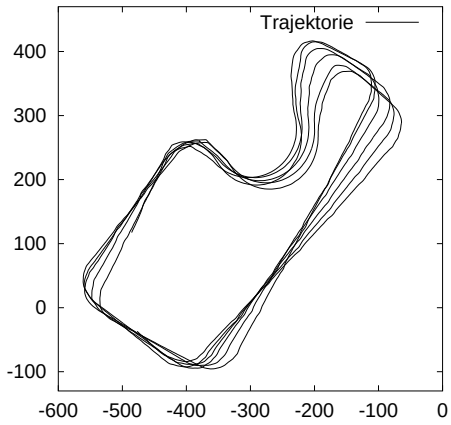


Abbildung 4.2: Weg des Roboters

Rundfahrt des Roboters (Trajektorie Abbildung 4.2) gezeigt. In die Roboterumgebung wurden vier unterschiedlich lange Kanten, ein Innen- und ein Außenradius eingebaut. Jede Nadelspitze beim Wert der Information markiert eine Richtungsänderung des Roboters. Der Wert der Information erreicht seinen tiefsten Wert beim Entlangfahren an der längsten Kante.

4.4 Anwendung in der Praxis

4.4.1 Das Umschaltproblem

4.4.1.1 Verzögerter Anfang des Updates

Wird der Update der Differentialquotienten nicht in einem Schritt vorgenommen, und das ist der Fall, wenn der Faktor ε aus Gleichung 4.9 kleiner als $1,0$ ist, dann erfolgt eine Mittelung über mehrere Zeitschritte. Ebenso erfahren die Wahrscheinlichkeiten p_i aus Gleichung 4.8 durch den Faktor η eine Mittelung. Das führt dazu, daß die Wahrscheinlichkeiten erst nach einigen Schritten ihr Maximum erreichen können. Der Update und damit die Verringerung des Fehlers und Erhöhung der Fitness erfolgen aber nur optimal, wenn die Wahrscheinlichkeit p über mehrere Schritte einen Wert in der Umgebung von 1.0 aufweist.

Nach Gleichung 4.8 ist die Summe aller Wahrscheinlichkeiten gleich $1,0$, d.h. wenn eine Gleichung eine Wahrscheinlichkeit von 95% hat, müssen sich alle anderen die restlichen 5% teilen. Die Wahrscheinlichkeit für eine unpassende Gleichung ist also sehr klein. Schwenkt der Roboter auf seiner Fahrt jetzt auf eine Bahn, deren zugehörige Gleichung noch eine niedrige Wahrscheinlichkeit hat, kann es sehr lange dauern, bis die neue, eigentlich passende Differentialgleichung ein Update erfährt. Bei sehr kurzen Wegstücken kann es sogar dazu kommen, daß überhaupt kein Update durchgeführt wird, weil dieses Wegstück bereits wieder verlassen wurde, bevor die Fisher-Eigen-Dynamik umschlagen konnte.

4.4.1.2 Schranke für p_i

Um diese Verzögerung klein zu halten, wird eine untere Schranke für die Wahrscheinlichkeiten definiert, die von keinem p_i unterschritten werden soll. Ist das doch der Fall, wird das entsprechende p_i gleich dem Wert dieser Schranke gesetzt. Da die Summe aller Wahrscheinlichkeiten p_i gleich $1,0$ sein soll ergibt sich eine obere Schranke, die die p_i nicht überschreiten können.

$$p_o = 1,0 - (N-1) * p_u \qquad p_u \approx 0,02 \tag{4.12}$$

Diese Schranken müssen beim Update der Wahrscheinlichkeiten p_i nach Gleichung 4.8 berücksichtigt werden. Falls eine Korrektur vorgenommen wurde, erhält das größte p_i Wert $p_i = 1 - \sum_j p_j,\ j \neq i$.

4.4.1.3 Verzögerter Update

Die Änderungen der Wahrscheinlichkeiten p_i folgen im Falle eines Umschwenkens immer den Fitnesswerten nach. In ungünstigen Situationen kann es passieren, daß die falschen Koeffizienten aufgrund ihrer noch hohen Wahrscheinlichkeitswerte einen Update mit völlig unpassenden Werten erfahren. Um das zu verhindern, können

1. die Lernparameter etwas vergrößert werden , damit schneller gelernt wird, oder
2. der Update kann mit verzögerten Werten von Δx und Δy vorgenommen werden, während die Fitness aus aktuellen Werten errechnet wird. Das Lernen der Koeffizienten findet damit ebenfalls verzögert statt.

Der erste Ansatz löst das Problem nicht prinzipiell, weil die Lernrate η indirekt die Länge des Wegstückes bestimmt, das mit einer bestimmten Kurve (Differentialgleichung) verglichen werden soll. Je höher die Lernrate gewählt wird, umso kürzer ist das "passende" Wegstück.

Der zweite Ansatz funktioniert nur unter der Annahme, daß der Roboter sich größtenteils auf geradlinigen Bahnen bewegt und ein "Einschwenken" auf eine andere Richtung relativ schnell erfolgt. Weiterhin muß nun doch ein Teilstück des zurückgelegten Weges mit Hilfe von Punktkoordinaten abgespeichert werden. Die gewählte Verzögerung muß der Weglänge entsprechen, die während der Richtungsänderung zurückgelegt wird. Der Betrag der Verzögerung kann durch Auswerten der Information nach Shannon über den Fitnesswerten Φ_i und den Wahrscheinlichkeiten p_i relativ gut geschätzt werden.

4.4.2 Spezielle Fehlerfunktion für Geraden

Für die Fehlerfunktion wird ein Maß für den Abstand zweier Differentialgleichungen[1] benötigt. Die aus dem gerade zurückgelegten Wegstück angenommene Gleichung soll mit jeder anderen Differentialgleichung verglichen werden. Der größte erreichbare Abstand soll per Definition den Wert $1,0$ erhalten. Da in diesem Spezialfall den Anstieg der Geraden in zwei Variablen $(a_{x,i}, a_{y,i})$ ausgedrückt wird, kann aus deren Vorzeichen auch noch eine Richtung (Vektor) bestimmt werden. Der Abstand soll den Wert $1,0$ haben, wenn die zwei Geraden den größten Abstand voneinander aufweisen, d.h. die Vektoren genau in die entgegengesetzte Richtung zeigen bzw. einen Winkel von $180°$ einschließen. Bei Identität soll der Abstand $0,0$ betragen.

[1] In diesem Spezialfall kann es der Winkel zwischen zwei Geraden sein.

Die Argumente für die Abstandsfunktion sind die Wegstrecken-Informationen $\Delta x_t, \Delta y_t$ (oder der Winkel α_t und die Weglänge Δs_t) aus einem Zeitschritt t und die zu lernenden Werte $a_{x,i}, a_{y,i}$, die den Winkel β_i (Ausrichtung) der Gerade i (Kante) im internen Koordinatensystem repräsentieren. Im ideal gelernten Zustand im Zeitschritt t sind die folgenden Ausdrücke äquivalent.

$$\tan(\alpha_t) = \frac{\Delta y_t}{\Delta x_t}, \quad \sqrt{\Delta x_t^2 + \Delta y_t^2} = \Delta s_t \tag{4.13}$$

und

$$\tan(\beta_i) = \frac{a_{y,i}}{a_{x,i}}, \quad \sqrt{a_{x,i}^2 + a_{y,i}^2} = 1 \tag{4.14}$$

Es wurden mehrere Abstandsfunktionen getestet.

1. Die erste betrachtete Abstandsfunktion ist die Fehlerfunktion nach Gleichung 4.5

2. In der zweiten Version wurde das Quadrat aus Gleichung 4.5 durch den Betrag ersetzt.
$$E = j(\alpha_t, \beta_i) = \frac{\left|\frac{\Delta x_t}{\Delta s_t} - a_{x,i}\right| + \left|\frac{\Delta y_t}{\Delta s_t} - a_{y,i}\right|}{2\sqrt{2}} \tag{4.15}$$

3. Die dritte Version der Abstandsfunktion widerspiegelt den Betrag des Winkelabstandes.
$$E = g(\alpha_t, \beta_i) = \arccos\left(\frac{\Delta x_t * a_{x,i} + \Delta y_t * a_{y,i}}{\Delta s_t}\right) * \pi^{-1} \tag{4.16}$$

In Abbildung 4.3 auf Seite 37 ist der Verlauf der einzelnen Funktionen an der Stelle $\beta = \frac{\pi}{5}$ dargestellt, der Wert für α läuft zwischen $-\pi$ und π. Alle Abstandsfunktionen haben den Maximalwert $1,0$, wenn die zu vergleichenden Vektoren in die entgegengesetzte Richtung zeigen. Das sehr langsame Ansteigen des Abstandes nach Gleichung 4.5 bei kleinen Winkeldifferenzen hat zur Folge, daß die Wahrscheinlichkeitswerte der einzelnen Differentialgleichungen nicht entsprechend eingestellt werden und die Information geringer ist (d.h. der Wert der Information nach Gleichung 4.11 steigt an). Für den Fall, daß mehrere Geraden nahe beieinander liegen, können Effekte wie das Heranziehen eigentlich unpassender Geraden auftreten. Bei der Funktion nach Gleichung 4.15 treten diese Effekte nicht mehr auf. Sie weist die größte Trennschärfe

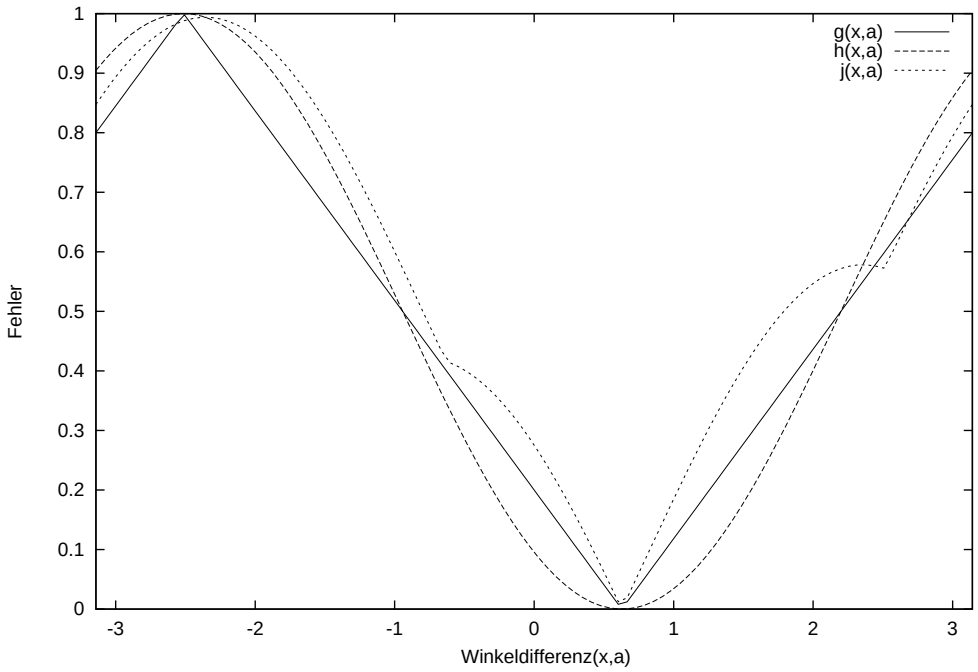


Abbildung 4.3: Abstandsfunktionen für Geradengleichungen

im Nahbereich auf, d.h. der Unterschied zwischen zwei ähnlich gut passenden Geradengleichungen ist größer, wodurch sich die Wahrscheinlichkeitswerte schneller neu einstellen können. Der gegenüber den Gleichungen 4.16 und 4.5 sehr ungewöhnliche Funktionsverlauf kann in Kauf genommen werden, da die Unregelmäßigkeiten nur in dem Bereich bemerkbar werden, in dem der Abstand zweier Geraden größer als 90° ist. Unter der Annahme, daß mindestens vier verschiedene Geradengleichungen der Menge der betrachteten Gleichungen angehören und der Raum, in dem sich der Roboter bewegt, vier rechtwinklig zueinander stehende Wände hat, so ist dieser Abstand gleichzeitig der größte erreichbare Abstand überhaupt.

Die Abstandsfunktion nach Gleichung 4.16 gibt mit der direkten Differenz zwischen zwei Winkeln zwar ein begrifflich sehr gut faßbares Abstandsmaß, benötigt aber den meisten Rechenaufwand. Es lassen sich außerdem Funktionen finden, die steilere Anstiege aufweisen. Im allgemeinen eignen sich Funktionen mit steilen Anstiegen in der näheren Umgebung des Referenzpunktes besser als Abstandsfunktion.

Die Fehlerfunktion nach Gleichung 4.5 ist im Prinzip eine zwischen $0,0$ und $1,0$ normierte Kosinusfunktion, die als Argument den Differenzwinkel der beiden Geraden (Vektoren) hat. Bei der Suche nach Funktionen mit steileren Anstiegen im Nahbereich fiel die Funktion $d = \frac{\cos(x)^3+1}{2}$ auf. Ein weiterer Vorteil dieser Funktion ist das Plateau bei $x = \pm\frac{\pi}{2}$, das bewirkt, daß alle in der näheren Umgebung liegenden Vektoren annähernd gleich stark berücksichtigt werden und die Entscheidung über den Updatesieger stärker durch die Wahrscheinlichkeiten p_i erfolgt.

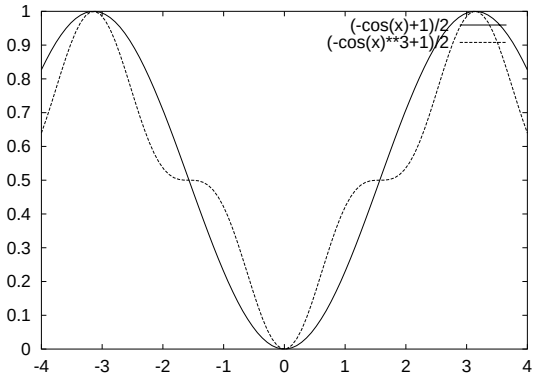


Abbildung 4.4: Abstandsfunktion mit steilerem Anstieg

$$E = \frac{1}{2} * \left(\cos\left(\left(\arctan\left(a_{y,i}, a_{x,i}\right) - \arctan\left(\Delta y_t, \Delta x_t\right)\right) - \pi\right)^3 + 1\right) \tag{4.17}$$

Diese Abstandsfunktion lieferte bei unseren Experimenten die besten Ergebnisse. Bei Richtungsänderungen des Roboters schlugen die Wahrscheinlichkeiten schnell um, und die Anzahl der Fälle, in denen zwei Differentialgleichungsobjekte über einen längeren Zeitraum gleiche Wahrscheinlichkeiten aufwiesen, wurde minimiert.

4.5 Erweiterungen

4.5.1 Kreisbahnsegmente

Nachdem gezeigt wurde, daß das Prinzip der lernenden Differentialgleichungen für einfache Kurvenformen anwendbar ist, kann zu komplizierteren Kurvenformen übergegangen werden. Die einfache Gleichung für den Kreisumfang $U = 2\pi R$ kann nach R umgestellt werden. Der Radius ergibt sich aus dem zurückgelegten Weg auf der Kreisbahn geteilt durch die Winkeldifferenz in der Ausrichtung.

$$R = \frac{U}{2\pi} \approx \frac{\Delta s}{\Delta \alpha} = a_i \tag{4.18}$$

Der Radius einer Kreisbahn ist also der Parameter a_i, der mit dieser Differentialglei-

chung gelernt werden kann. Durch negative Winkel kann der Parameter a_i auch ein negatives Vorzeichen bekommen. Kreisradien sind aber immer positiv, das Vorzeichen bezeichnet hier die Richtung, auf der sich der Roboter auf der Kreisbahn bewegt (positiv = entgegen dem Uhrzeigersinn). Die Werte von $\Delta\alpha$ und Δs können sehr stark fluktuieren. Das hat zur Folge, daß der Parameter a, der ja der Quotient aus $\Delta\alpha$ und Δs ist, noch stärkeren Schwankungen unterliegen würde. Um Fehlern bei der Berechnung im Computer entgegenzuwirken, werden $\Delta\alpha$ und Δs als getrennte Variablen im Speicher gehalten. Der Update der Kreisgleichungsparameter erfolgt (analog zu Gleichung 4.9) mit dieser Gleichung:

$$\Delta a_i = \varepsilon * p_i * (a_i * \Delta\alpha - \Delta s) \tag{4.19}$$

4.5.2 Abstandsfunktion für Kreisradien

Die Fehlerfunktion soll den Abstand von zwei Kreisradien auf den Wertebereich $[0\,..\,1,0]$ abbilden.

$$E_i = 1 - \frac{\left(\frac{k}{2}\right)^2}{\left(a_i * \Delta\alpha - \Delta s\right)^2 + \left(\frac{k}{2}\right)^2} \quad k\epsilon[1\,..\,1000] \tag{4.20}$$

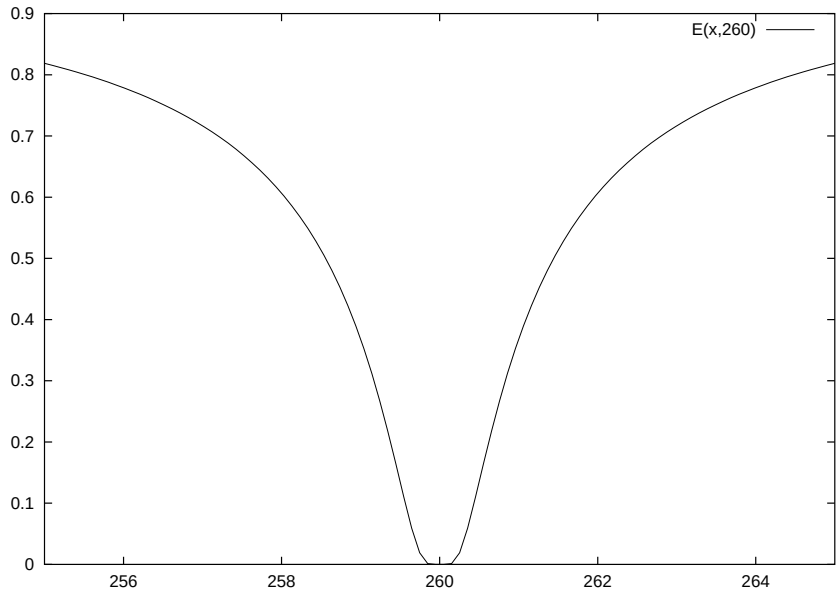


Abbildung 4.5: Abstandsfunktion für Kreisradien

Mit dem Parameter k läßt sich die Steilheit bzw. mittlere Breite der Funktion einstellen und damit die Trennschärfe je nach Anwendungsgebiet beinflussen oder geeignet auswählen. Im Unendlichen steigt der Wert der Funktion jeweils auf $1,0$.

4.5.3 Kreisgleichungen gegen Geradengleichungen

Werden Versuche mit verschiedenen Arten von Differentialgleichungsobjekten in einem Pool Ω (siehe 4.2.2) durchgeführt, ist festzustellen, daß eine Art nur in besonders wenigen Fällen Updates gewinnt.

So können sich zum Beispiel Differentialgleichungsobjekte für Kreisbahnen nur dann gegen die für Geraden durchsetzen, wenn wenige andere Differentialgleichungsobjekte vorhanden sind. Anderenfalls ist der Konkurrenzdruck zu hoch und die Wahrscheinlichkeiten können sich nicht auf einem Wert stabilisieren, bei dem Updates möglich werden. Durch die Wahl eines größeren Wertes für k in Gleichung 4.20 kann zwar der Einzugsbereich für bestimmte Instanzen von Differentialgleichungsobjekten vergrößert werden, beschränkt aber gleichzeitig die Anzahl der sinnvoll nutzbaren Objekte. Ein zu großer Wert für k führt allerdings dazu, daß diese eine Gleichung immer wieder einen sehr hohen Fitnesswert aufweist und damit kein Umschlagen der Wahrscheinlichkeiten erfolgen kann.

Allgemein kann gesagt werden, daß kompliziertere Kurvenformen seltener einen Update gewinnen, weil die einfacheren Kurvenformen (z.B. Geraden) in ihnen schon enthalten sind. Damit weist neben der betrachteten komplizierten Kurvenform auch immer eine einfache Kurvenform eine relativ hohe Fitness auf, und die Information nach Gleichung 4.11 bleibt unter dem Schwellwert für zulässige Updates. Da sich die Konstruktion von wirklich gleichwertigen Abstandsfunktionen für verschiedene Klassen von Differentialgleichungsobjekten schwierig gestaltet, werden immer nur gleichartige Objekte in einer Menge Ω zusammengefaßt.

4.6 Virtuelle Rotation und Driftkorrektur

4.6.1 Typische Fehlerbilder

In Abbildung 4.6a ist der zurückgelegte Weg des Roboters bei einer Versuchsfahrt dargestellt. Ein Skalenteil entspricht dabei einem Millimeter in der wirklichen Welt. Der Roboter wurde auf Wandverfolgung programmiert und hat die aufgebaute Welt

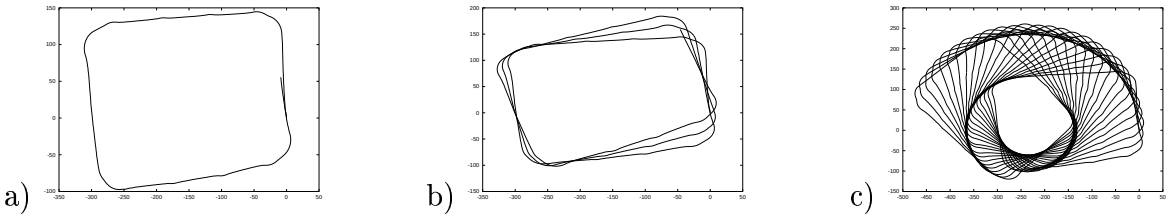


Abbildung 4.6: Trajektorie des Roboters nach 1, 3 und 21 Umrundungen, Rechteck-Welt

(ein Rechteck mit abgerundeten Ecken) 21 mal in einer Richtung umrundet. Dabei wurde nur aus den Odometriedaten der Räder die aktuelle Position des Roboters ca. aller 1, 2 mm zurückgelegten Weges neu berechnet. Deutlich ist ein systematischer Fehler bei der Berechnung der aktuellen Position zu erkennen, der von verschiedenen Reibwerten der Räder oder Differenzen der Raddurchmesser herrühren kann. Durch diesen Fehler wird die momentane Ausrichtung des Roboters falsch berechnet und erzeugt eine virtuelle Drehung des Weltkoordinatensystems. Die Winkelgeschwindigkeit[2] dieser Drehung wird nur durch Eigenschaften des Roboters (Raddurchmesser, Radstand, Spurweite, Bereifung) und Eigenschaften des Untergrundes bestimmt. Die Abhängigkeit dieses Fehlers von der Länge oder Art des zurückgelegten Weges oder der Gesamtdrehung des Roboters bezüglich seiner ursprünglichen Ausrichtung ist nicht durch eine vorher bekannte Funktion darstellbar. Die in Abbildung 4.7 dargestellte Trajektorie macht das deutlich. Sie wurde unter den gleichen Bedingungen wie die

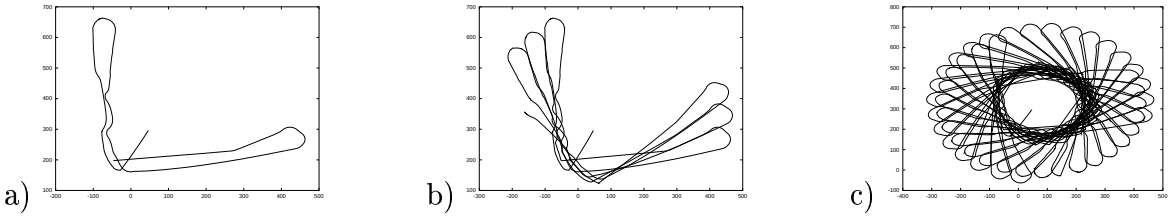


Abbildung 4.7: Trajektorie des Roboters nach 1, 3 und 24 Umrundungen, L-Welt

erste Trajektorie aufgenommen. Im Gegensatz zu der in Bild 4.6 gezeigten Trajektorie beträgt die Gesamtdrehung des Roboters in diesem Fall nach jeder durchlaufenen Runde wieder 0°. Die Bahn ist im Prinzip eine geknickte, liegende Acht. Der Roboter mußte auf seinem Weg im selben Maß Links- und Rechtsdrehungen ausführen. Die Winkelabweichung war in diesem Fall kleiner, bleibt aber trotzdem ein systematischer Störfaktor. Auf das Phänomen der virtuellen Rotation der Weltkoordinaten durch fehlerbehaftete Positionsbestimmung wird auch in [Gut96] hingewiesen.

[2]Winkelabweichung pro Umlauf bzw. pro zurückgelegter Wegstrecke

4.6.2 Driftkorrektur

Durch eine Erweiterung des oben beschriebenen Algorithmus kann zusätzlich zu den gelernten Bahnparametern ein Driftparameter γ gelernt werden, der die Winkelgeschwindigkeit der virtuellen Rotation des Weltkoordinatensystems widerspiegelt. Durch Integration von γ kann zu jedem Zeitpunkt die aktuelle Verdrehung φ des Koordinatensystems bestimmt und damit der systematische Fehler korrigiert werden. Die Parameter γ und φ werden in jedem Zeitschritt neu berechnet.

$$\varphi_{neu} = \varphi_{alt} + \gamma \tag{4.21}$$

und

$$\gamma_{neu} = \gamma_{alt} + \Delta\gamma \tag{4.22}$$

Um die Änderung der Winkelgeschwindigkeit $\Delta\gamma$ erfassen zu können, werden gut angepaßte Geradengleichungen benötigt. Damit diese Anpassung durch die permanente virtuelle Drehung der Weltkoordinaten nicht verlorengeht, wird der Update der Geradengleichungen nach Gleichung 4.9 schon mit den rückgedrehten Werten δx_s und δy_s anstelle von Δx und Δy durchgeführt.

$$\overrightarrow{r} = \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}, \quad T = \begin{pmatrix} \cos(-\varphi) & \sin(-\varphi) \\ -\sin(-\varphi) & \cos(-\varphi) \end{pmatrix}$$

$$\begin{pmatrix} \Delta x_t \\ \Delta y_t \end{pmatrix} = \overrightarrow{r_t} = T\overrightarrow{r} \tag{4.23}$$

$$\Delta\gamma = \sum_{i=1}^{n} \frac{\varepsilon\sigma(p_i)\Delta s}{K} * (((\delta x_s - a_{x,i}) * \delta y_s) - ((\delta y_s - a_{y,i}) * \delta x_s)) \tag{4.24}$$

$$\delta x_s = \frac{\Delta x_t}{\Delta s}, \quad \delta y_s = \frac{\Delta y_t}{\Delta s}, \quad K\epsilon[5000\,..\,50000]$$

Mit dem Faktor K wird das Verhältnis der Lerngeschwindigkeit der einzelnen Koeffizienten a_i zur Winkelgeschwindigkeit $\Delta\gamma$ eingestellt. Für ε wird der gleiche Wert wie in Gleichung 4.9 angesetzt. In den Abbildungen 4.8 und 4.9 sind die in Echtzeit korrigierten Trajektorien der oben gezeigten Testläufe zu sehen. Die restliche Parallelverschiebung resultiert aus der Differenz von angenommenem und eigentlichem Mittelpunkt des Roboterkoordinatensystems während der Rückdrehung der Punkte

einer Trajektorie. Da die Parameter für den Verdrehungsfehler online gelernt werden,

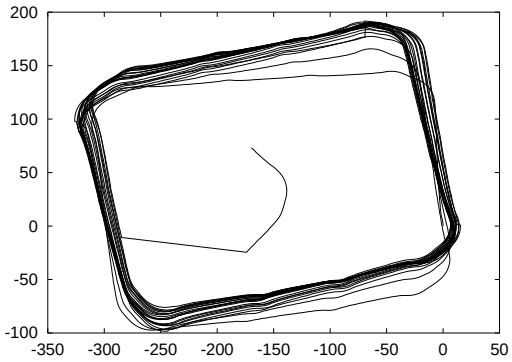


Abbildung 4.8: online korrigierte Version der Trajektorie aus Abbildung 4.6

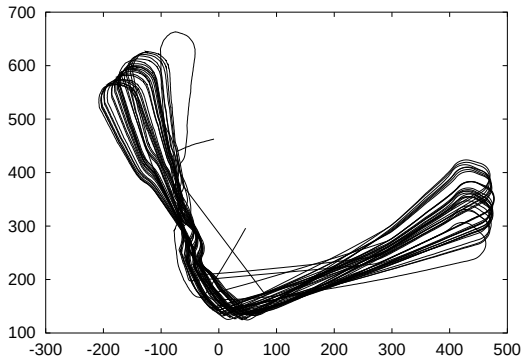


Abbildung 4.9: online korrigierte Version der Trajektorie aus Abbildung 4.7

können auch Änderungen des Untergrundes, die eine Änderung des systematischen Fehlers hervorrufen, automatisch korrigiert werden.

4.7 Ausblick

Der vorgestellte Algorithmus hat gezeigt, daß er zur Landmarkenerkennung, speziell zur Erkennung von bestimmten Pfadprofilen, genutzt werden kann. Er ist dabei nicht an das Verfahren der Odometrie gebunden. Die Strukturen der Umgebung (Wände) müssen nicht durch Wandverfolgung "erlaufen" werden, sondern können auch das Ergebnis eines Laserscans sein. Werden die Punkte eines Laserscans nach dem Winkel, unter dem sie aufgenommen wurden, sortiert[3], ergibt sich annähernd dieselbe Trajektorie wie bei Wandverfolgung[4]. Mit mehreren aufeinanderfolgenden Scans (auch aus der Bewegung heraus) kann die Orientierung im Raum direkt und kontinuierlich bestimmt werden und man erhält eine Art Kompaß, der sich an den Gegebenheiten des Raumes (z.B. Wände) orientiert.

In [Gut96], Abschnitt 5.3 beschreibt J. S. Gutmann einen Algorithmus zur Berechnung der Verdrehung eines Laserscans unter Benutzung eines Winkelhistogramms. Der oben beschriebene Algorithmus erzielt in ähnlicher Weise die gleichen Ergebnisse, nur mit dem Unterschied, daß die Winkel direkt aus den Koeffizienten der Differentialgleichungsobjekte abgelesen werden können, ohne daß eine Kreuzkorrelation benötigt

[3] dies ist bereits während des Scanvorgangs möglich
[4] Verdeckte Wände sind nicht berücksichtigt

wird. Weiterhin sind erste (wenn auch nur partiell korrekte) Ergebnisse schon nach einigen wenigen Updateschritten verfügbar, und eine Erhöhung der Anzahl der Scanpunkte verbessert auch die Genauigkeit, ohne daß signifikant mehr Rechenleistung benötigt wird. Da in jedem Schritt immer dieselben einfachen Operationen durchgeführt werden, kann dieser Algorithmus auch direkt in Hardware realisiert werden.

Kapitel 5

Anwendung 2
Das Prinzip der *homeokinesis*

In diesem Kapitel wird erläutert, daß grundlegende Verhaltensweisen von autonomen Agenten ohne die Mithilfe von außen definierter Fitnessfunktionen oder Lernziele erworben werden können. Es wird das Prinzip der *homeokinesis* vorgestellt, das sehr allgemein formuliert ist und trotzdem sehr spezifische, anscheinend zielorientierte Verhaltensweisen eines Agenten in einer komplexeren Umwelt hervorbringt. Das Prinzip basiert auf der Annahme, daß der Agent ein internes Selbstmodell seines aktuellen Verhaltens erlernen und dieses Verhalten so verändern kann, daß die Lücke in der Komplexität seines Verhaltens im Vergleich zu seinem Selbstmodell nicht zu groß wird. Angetrieben von der Maxime sich immer zu bewegen und zu agieren, entwickelt der Agent in einer komplexen Umgebung mit verrauschten Sensordaten ein weiches Steuerverhalten als Kompromiß zwischen der hohen dynamischen Komplexität der Umgebung und der niedrigen einfachen Komplexität des Selbstmodells von der Welt. In diesem Sinne kann das Verhalten des Agenten als Nebenprodukt seines Wunsches, die Umgebung in einfachen Ausdrücken zu erfassen, angesehen werden.

5.1 Einleitung

Bei dem verhaltensbasierten Ansatz in der Robotik besteht ein großes Interesse, das Phänomen der emergenten Verhaltensweisen besser zu verstehen. Der geradlinigste Weg dahin ist die Anwendung künstlicher Evolution zur Entwicklung von Verhaltensformen, die sich an Fitnessfunktionen orientieren. Ein anderer Weg ist die Anwendung

von *reinforcement learning*, bei dem Verhaltensmuster durch das gezielte Vergeben von Belohnungen in eine bestimmte Richtung entwickelt werden. Trotz einiger sehr interessanter Erfolge gibt es auch kritische Stimmen über die erzielten Resultate. Alle bisher durch Evolution hervorgebrachten Verhalten sind sehr einfach und basieren mit wenigen Ausnahmen [NF97],[NML95] auf Computersimulationen. Der Hauptgrund hierfür liegt in der Komplexität der Aufgabe, so daß *reinforcement learning* nur sehr langsam konvergiert oder Evolution über viele Generationen laufen muß. Die Lösung dieser Schwierigkeiten könnte eine Beschleunigung dieser Verfahren sein. Die Erfahrungen, die bei der Entwicklung spezieller Controller für den *khepera*-Roboter gesammelt wurden, haben gezeigt, daß das Erzeugen bestimmter Verhaltensweisen auf ein grundsätzliches Problem, unabhängig vom verwendeten Ansatz, trifft.

5.2 Das Design-Problem

In der künstlichen Evolution ist es die Gestaltung der Fitnessfunktion, die im Hinblick auf das zu erzeugende Verhalten "optimiert" wird. Nicht selten kommt dabei ein ganz anderes Verhalten heraus, als geplant war. Wenn *reinforcement learning* zum Einsatz kommt, ist es die günstige Verteilung der Belohnungen, die entscheidend für den Erfolg ist.

Ein gemeinsames Problem aller Verfahren zur Erzeugung von Verhaltensweisen ist, daß Zwischenziele sehr sorgfältig gewählt werden müssen, um die Entwicklung in eine bestimmte Richtung voranzutreiben. Die Evolution verlief in der Natur nicht auf diese Art und Weise. Beobachtet wird die Entwicklung von sehr zielorientiert erscheinenden Verhaltensweisen, die aus völlig unspezifischen Grundprinzipien entstehen (z.B. bei der Entstehung eines Termitenhügels, siehe [Wil71]).

Das bekannteste Prinzip auf diesem Gebiet ist *homeostasis*, das von CANNON [Can39] schon 1939 auf dem Gebiet der Physiologie eingeführt wurde. Er schloß aus seinen Beobachtungen, daß die komplizierten, miteinander wechselwirkenden Regelkreise im Körper nur physiologische Größen wie z.B. Blutdruck oder Blutzuckergehalt auf konstante Pegel regeln und die Reaktionen des Körpers nur ein äußerlich sichtbares Abprodukt dieser internen Regelung sind. Dieses Prinzip ist allgemein und nicht an einen spezifischen Prozeß gebunden.

Bisher wurden wenig Anwendungen für das Prinzip der *homeostasis* gefunden. In [DSP99] und [DP99] wird das Prinzip der *homeokinesis* als dynamisches Pendant zur *homeostasis* eingeführt. Das interne Ziel des Agenten ist hierbei nicht das Erreichen

eines bestimmten statischen Zustandes, sondern das Beibehalten eines internen kinetischen Regimes. Dieses interne Regime wird mit Hilfe eines *anpassungsfähigen, internen Selbstmodells* des Agenten formuliert. Es wird gezeigt, daß die Diskrepanz von Selbstmodell und realem Verhalten als immer zur Verfügung stehendes Lernsignal benutzt werden kann, um verschiedene Verhaltensweisen hervorzubringen.

5.3 Das Selbstmodell

Der wichtigste Aspekt der beiden Prinzipien *homeokinesis* und *homeostasis* ist die Betonung der inneren Sichtweise. Verhalten wird nicht durch extern festgelegte Ziele erzeugt, sondern entwickelt sich aus dem inneren Antrieb des Agenten. Das Weltbild des Agenten wird aus der inneren Sichtweise durch die Sensorwerte vermittelt.

In Abbildung 5.1 ist das Funktionsmodell eines homöokinetischen Controllers dar-

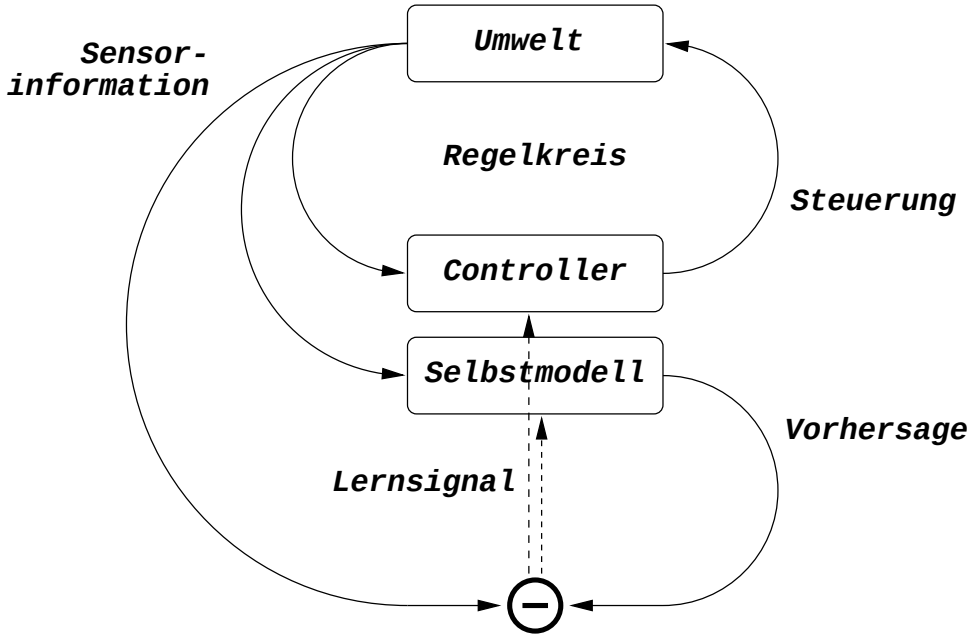


Abbildung 5.1: Funktionsmodell des *homeokinesis*-Controllers

gestellt. Die Steuereinheit bildet mit der Umwelt einen einfachen Regelkreis. Parallel dazu werden die Sensorinformationen in ein Selbstmodell eingegeben. Dieses Selbstmodell hat gelernt, die erwarteten zukünftigen Sensorwerte vorherzusagen. Die Ausführung dieser Vorhersage dauert etwas, sodaß die Ausgabe mit den neuen Sensorwerten zeitlich zusammenfällt. Aus der Differenz bzw. dem Fehler von vorhergesagten und eingetretenen Sensorwerten wird ein Lernsignal abgeleitet, mit dem sowohl

das Selbstmodell als auch der Controller trainiert werden. Die Steuerung wird so verändert, daß der Fehler minimiert wird. Die Komplexität des Selbstmodells bestimmt die Art des erzeugten Verhaltens.

5.4 Interne Repräsentation

Der Controller des Roboters ist eine Funktion, die den Vektor der Sensordaten in einer bestimmten Situation auf eine Aktion abbildet. Diese Aktion bewirkt eine neue Situation und erzeugt wiederum neue Sensordaten. Jeder Controller produziert auf diese Weise eine spezielle Trajektorie $x\,(t)$ im Sensorraum. Angenommen der Roboter fährt mit konstanter Geschwindigkeit und sein Controller bestimmt nur die Drehgeschwindigkeit, mit der der Roboter Kurven fährt. Dann kann die Funktion des Controllers z.B. so aussehen:

$$\omega = g\left(\sum_{i=1}^{n} c_i x_i\right) \tag{5.1}$$

wobei für die verallgemeinerte Glättungsfunktion $g(...)$ zum Beispiel der *tangens hyperbolicus* verwendet werden kann. Die c_i seien passend gewählt und $x_i \epsilon [0,1] \forall i$. Die Sensorwerte sollen in regelmäßigen Intervallen ausgelesen werden, und δx sei die Änderung der Sensorwerte im aktuellen Zeitschritt. Dann ist die Dynamik im Sensorraum gegeben durch

$$\Delta x = F(x,c) \quad x \epsilon R^n, c \epsilon R^p \tag{5.2}$$

Die Funktion F kann als Generator einer zeitlichen Entwicklung angesehen werden

$$x\,(t+1) = x\,(t) + F\,(x\,(t)\,, c) \tag{5.3}$$

Die Gleichungen 5.2 bzw. 5.3 kann als Updateregel eines zeitdiskreten dynamischen Systems angesehen werden. Für hinreichend kleine Δx kann das Ganze auch als zeitkontinuierliches dynamisches System aufgefaßt werden[1].

$$\delta x = F\,(x,c) \tag{5.4}$$

Die Komplexität der Dynamik aus den Gleichungen 5.2 und 5.4 hängt für eine bestimmte Umgebung und fest gewählte Startbedingungen nur vom Controller ab.

[1] mit einer leicht abgewandelten Funktion $F(x,c)$

Mit einem Controller aus Gleichung 5.1 bewegt sich der Roboter so lange geradeaus, wie seine Sensoren nichts messen. Trifft er auf ein Hindernis, steigen die Sensorwerte an und bleiben konstant, wenn der Roboter an dem Hindernis steckengeblieben ist. Sind die Parameter $c_i \gg 1$, $\forall i$ und der Roboter bewegt sich in einem Labyrinth, ist die Trajektorie sehr chaotisch, weil der Roboter zusätzlich zu seiner Vorwärtsbewegung sich auch noch schnell dreht, solange die Sensorwerte verschieden von Null sind. Falls der Roboter im konstanten Abstand einer Wand folgt, bleiben die Sensorwerte konstant. Nur wenn der Controller ein stabiles Wandfolgeveralten erzeugt, kann der Roboter an jeder Stelle im Labyrinth gestartet werden. Das heißt, seine Sensorwerte konvergieren gegen $x = x_{Wand}$. x_{Wand} ist also ein Attraktor der Dynamik im Sensorraum. Auf die selbe Art und Weise wird die Verfolgung eines bewegten Objektes mit konstantem Abstand von einem Attraktor im Sensorraum gekennzeichnet.

Offensichtlich gibt es mehrere Verhaltensweisen, die durch einen Attraktor im Sensorraum gekennzeichnet sind. Man könnte das als Werkzeug nutzen, um ein sinnvolles Verhalten in einer gegebenen Welt anwendungsunabhängig zu definieren.

Angenommen ein Controller kann einen Roboter sehr elegant durch ein kompliziertes Labyrinth mit Türen, Korridoren und verschiedenen Hindernissen steuern. Dieser Controller erzeugt nicht nur einen Attraktor der Dynamik im Sensorraum, sondern er produziert auch Trajektorien mit moderater Komplexität zwischen den oben erwähnten chaotischen und trivialen Trajektorien.

Ein domäneninvariantes Prinzip zum Anlernen eines Controllers läßt sich damit ableiten. Das Problem ist, daß die Komplexität irgenwie spezifiziert sein muß. Das kann durch ein adaptives Selbstmodell realisiert werden.

$$\delta x^P = \Phi(x, u) \tag{5.5}$$

Die Änderung der Sensorwerte δx^P wird auf der Grundlage der aktuellen Sensorwerte x und des Parametersatzes u vorhergesagt. Die Funktion Φ definiert eine Dynamik im Sinne von Gleichung 5.3. Das heißt, das zeitdiskrete dynamische System

$$\hat{x}\,(t+1) = \hat{x}\,(t) + \Phi(\hat{x}, u)$$

definiert eine Zeitreihenentwicklung von $\hat{x}$, was ein Modell des Verhaltens des Roboters darstellt. Die Funktion Φ definiert das Selbstmodell des Roboters. Der Parametersatz u bestimmt die Komplexität des Dynamik. Bestimmte festgelegte u erzeugen Trajektorien mit einer bestimmten Komplexität. Die Menge aller Trajektorien, die von allen

gültigen Werten von u erzeugt werden können, wird als Komplexitätsklasse aufgefaßt und von der Struktur der Funktion Φ bestimmt. Aus der Sichtweise der *homeokinesis* bestimmt die Struktur der Funktion Φ ein kinetisches Regime, das der Agent zu erreichen und zu stabilisieren versucht.

Als Beispiel seien zwei Modelle gegeben:

$$\Phi = 0 \tag{5.6}$$

und

$$\Phi_i = w \sum_{j=0}^{n} u_{i,j} x_j \tag{5.7}$$

In Gleichung 5.6 wird angenommen, daß keine Änderung der Sensorwerte stattfindet. Die Struktur dieses Modells ist trivial. Gleichung 5.7 sagt aus, daß jede Änderung der Sensorwerte proportional zur Drehgeschwindigkeit des Roboters sein soll. Das sind sehr einfache Modelle, aber sie reichen aus, um verschiedene nichttriviale Verhaltensformen zu erzeugen.

In allen praktisch interessanten Fällen ist die dynamische Komplexität des Modells geringer als die des wirklichen Verhaltens. Das heißt, es existiert eine Lücke in der Komplexität zwischen Modell und Realität. Die Idee ist, die Größe dieser Lücke als Lernsignal für die Anpassung von Modell *und* Controller heranzuziehen. Das Ziel ist, den Controller so anzupassen, daß er möglichst gut in die von Φ definierte Komplexitätsklasse paßt. Dazu wird eine Fehlerfunktion definiert

$$E = \frac{1}{2} \sum D_i^2 \tag{5.8}$$

mit

$$D_i = \delta x_i - \delta x_i^P \tag{5.9}$$

Mit anderen Worten: Der Fehler resultiert aus der Differenz der vorhergesagten und der eingetretenen Sensorwertänderung. Die Änderung der Sensorwerte δx_i ist eine Funktion von der aktuellen Drehgeschwindigkeit w, die wiederum eine Funktion von x_i und den Controllerparametern c_i ist. Das heißt, $E = E(x, c, u)$. Mit dieser funktionalen Abhängigkeit kann die Updateregel für die Adaption der Controllerparameter c_i

$$\Delta c_i = -\varepsilon \frac{\vartheta}{\vartheta c_i} E \tag{5.10}$$

und die Adaption der Modellparameter u_i

$$\Delta u_i = -\varepsilon \frac{\vartheta}{\vartheta u_i} E \tag{5.11}$$

definiert werden. Die Gleichung 5.11 kann ausführlicher formuliert werden:

$$\Delta u_i = \varepsilon \sum_j D_j \frac{\vartheta}{\vartheta u_i} \Phi_j(x, u) \tag{5.12}$$

Die partiellen Ableitungen von Φ sind explizit bekannt. Auf die Probleme bei der Evaluierung der partiellen Ableitungen von c_i wird im späteren Text noch eingegangen.

Die Parametersätze c und u werden in jedem Zeitschritt gelernt. Damit steht immer ein Gradient zum Lernen des Verhaltens zur Verfügung, der *nicht* von einem externen Lernziel abhängig ist.

5.5 Die Antwort-Methode

Um den Controller anzulernen wird der Gradient des Fehlers E in Abhängigkeit der Controllerparameter c_i benötigt. Dieser Gradient ist nicht explizit bekannt, weil es einer Reaktion des Controllers in der Welt bedarf, um die Controllerparameter zu ändern. Die Ableitung aus Gleichung 5.10 wird folgendermaßen aufgesplittet

$$\frac{\vartheta E}{\vartheta c_i} = \frac{\vartheta E}{\vartheta w} \frac{\vartheta w}{\vartheta c_i} \tag{5.13}$$

Die letzte Ableitung kann explizit mit Hilfe von Gleichung 5.1 ausgerechnet werden. Die verbliebene Ableitung lautet $\frac{\vartheta E}{\vartheta w}$. Diese Gradienteninformation wird mit einem Trick gewonnen. Man addiert eine kleine Störung $\phi(t)$ auf die Ausgabe w des Controllers. Das kann ein Rauschen oder eine periodische Oszillation sein. Wichtig ist, daß die Störung über einen längeren Zeitraum gemittelt Null ergeben muß. Unter der Annahme, daß die Änderung von ϕ klein in einem Zeitschritt ist, kann die Fehlerfunktion E als Funktion von ϕ formuliert werden.

$$E(x, w + \gamma\phi, u) = E(x, w, u) + \gamma\phi \frac{\vartheta E}{\vartheta w} + ... \tag{5.14}$$

Die Funktion ϕ wird so gewählt, daß über einen längeres Intervall gilt

$$\overline{\phi} \ll \overline{\phi^2} \tag{5.15}$$

Dadurch wird

$$\overline{E\phi} \approx \gamma \overline{\phi^2} \frac{\vartheta E}{\vartheta w} \tag{5.16}$$

Die Gleichung 5.10 kann wie folgt formuliert werden

$$\Delta c_i = -\varepsilon \phi(t) E \frac{\vartheta w}{\vartheta c_i} \tag{5.17}$$

Die Bildung des Mittels wird implizit durch die Updateregel durchgeführt (kleine Werte für ε). Der restliche Teil der Ableitung kann direkt ausgerechnet werden. Für einen Controller nach Gleichung 5.1 gilt

$$\frac{\vartheta w}{\vartheta c_i} = x_i g'(x_i) \tag{5.18}$$

5.6 Experimente mit *khepera*-Robotern

Der obengenannte Algorithmus wurde auf dem *khepera*-Roboter implementiert. Der Controller wurde nach zwei verschiedenen Gleichungen implementiert. Zuerst wurde der Controller wie folgt gewählt:

$$w = g\left(\sum_{i=1}^{n} \theta_i (x_i - c_i) x_i\right) \tag{5.19}$$

Dabei ist n die Anzahl der Sensoren und θ_i ist gleich $1,0$ für Sensoren auf der rechten und $-1,0$ für Sensoren auf der linken Seite des Roboters. Diese Form wurde gewählt, um ein stabiles Wandfolgeverhalten zu produzieren. Später stellte sich heraus, daß das nicht notwendig ist und der Controller nach Gleichung 5.1 genauso gut und stabil das Wandfolgeverhalten erzeugen kann. Mit beiden Controllern läuft der Roboter geradeaus, wenn er sich im freien Raum befindet (Sensoren $x_i = 0$). In einem Experiment werden die Controllerparameter zum Start zufällig gewählt oder auf Null initialisiert. Der Roboter fährt typischerweise erst ein paar mal gegen die Wand und muß von Hand wieder in freies Gelände bewegt werden. Nach einigen Versuchen lernt der Roboter, die Wand zu meiden, indem er schnell abdreht, oder er folgt ihr sogar. Falls keine größeren Änderungen mehr auftreten (Vorsprünge oder Kanten in der Wand) ist das Wandfolgeverhalten auch über einen längeren Zeitraum stabil, wie die Abbildung 5.2 eines Langzeitversuchs über 3 Tage deutlich zeigt. Während des Langzeitversuchs wurde keine Änderung an den Parametern vorgenommen. Insbesondere die Lernrate ε wurde nicht "abgekühlt". Die Parameter c_i wurden zu Beginn des Versuchs auf

Null initialisiert, sodaß der Roboter am Anfang nur geradeaus lief. Ein interessantes Phänomen konnte nach kurzer Zeit beobachtet werden. In einer Umgebung ohne Kanten und Vorsprünge in den Wänden lernte der Roboter ein neues Verhaltensmuster. Er steuerte in einem sehr engen Distanzbereich mit zwei Sensoren, so daß der der Wand am nächsten liegende Sensor ein Maximum lieferte und der zweitnächste Sensor ungefähr halben Ausschlag zeigte. Dieser nichttriviale Effekt kann als der Versuch der Minimierung des Einflusses des Rauschens angesehen werden, wie er in [DSP99] beschrieben wurde.

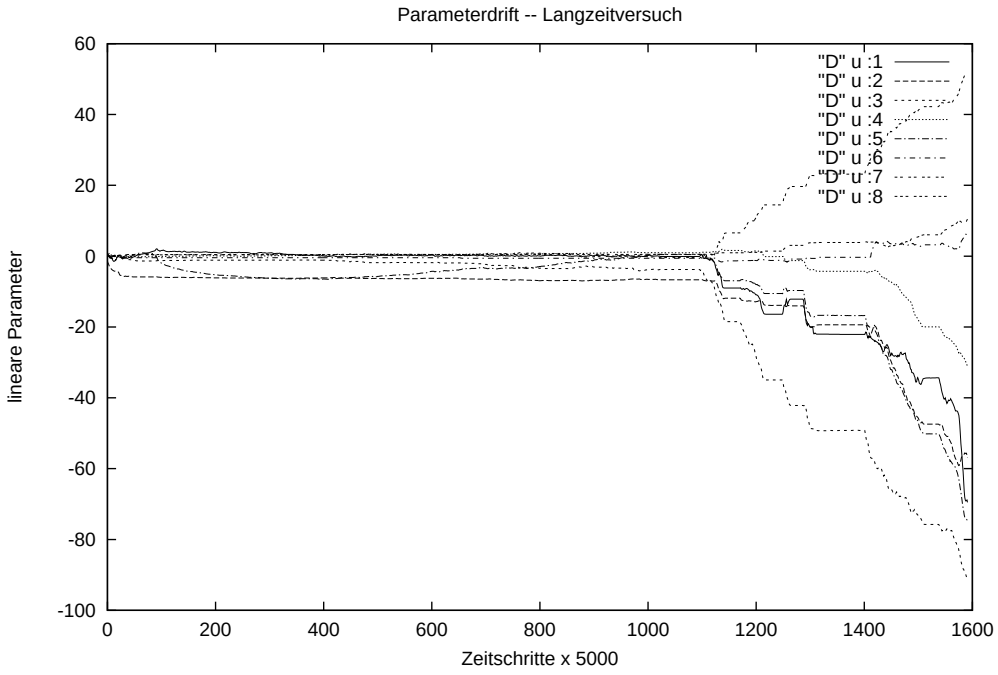


Abbildung 5.2: Controllerparameter bei Langzeitversuch

Die Abbildung 5.2 zeigt die Entwicklung der Controllerparameter c_i, $i = 1..8$ über einen längeren Zeitraum. Der *khepera*-Roboter lief ca. 8 km in einem Gatter mit einer Geschwindigkeit von 30mm/s. Die Parameter c_i für die beiden Frontsensoren wurden auf $2,2$, alle anderen zufällig zwischen $0,0$ und $1,0$ initialisiert. Nach einer anfänglichen Lernphase (Abbildung 5.3) hielt sich der Roboter für lange Zeit (ca. 2 Tage) stabil im Wandfolgemodus. Bedingt durch eine Veränderung der Umweltbedingungen, ein Wandteil wurde vom Roboter selbst ungünstig verschoben, wechselte der Roboter sein Verhalten und begann, sich in einer Ecke zu drehen. Während der anfänglichen Wandfolge-Phase lernte der Roboter nur die Parameter zu den Sensoren auf seiner linken Seite, da nur diese Sensoren in dieser Phase Input lieferten. Der Zeitpunkt des Wechsels in das Rotationsverhalten kann im Diagramm anhand der

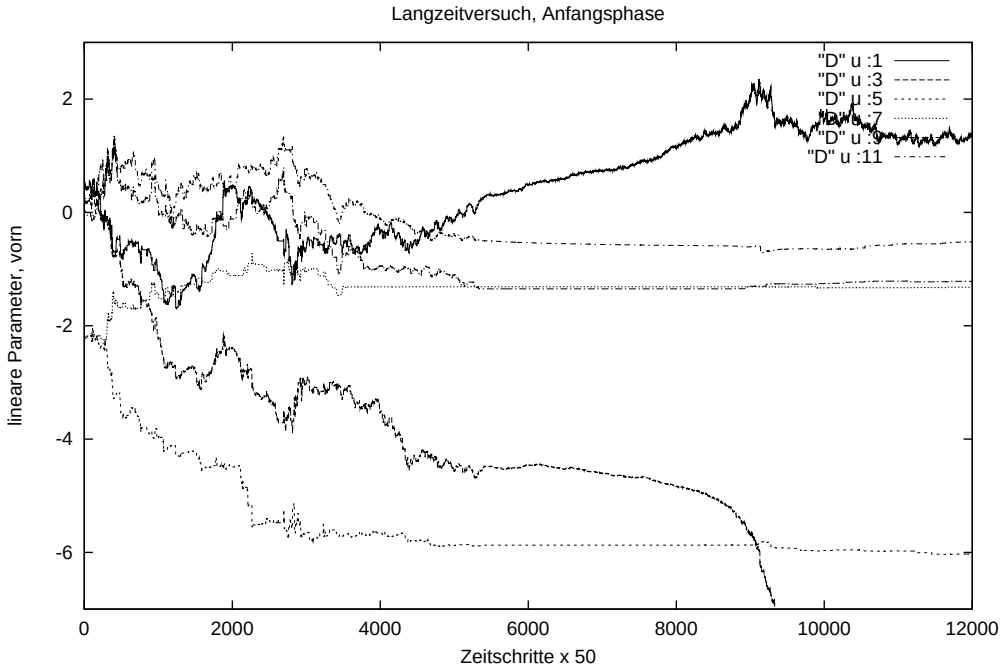


Abbildung 5.3: Anfangsphase des Langzeitversuchs

auseinanderstrebenden Parameter nach ungefähr $5,5$ Millionen Zeitschritten erkannt werden.

In Abbildung 5.4 ist die Entwicklung der Ballverfolgung zu erkennen. Dem Roboter wurde in der Lernphase ein Tischtennisball vor die "Nase" gehalten. Nach einiger Zeit lernte er, einem wegrollenden Ball hinterherzulaufen, solange seine Geschwindigkeit zur Verfolgung ausreichte.

Ein weiterer interessanter Aspekt ist, daß der Roboter sein Verhalten in Abhängigkeit von den Umgebungsbedingungen ändern kann. Hat der Roboter z.B. Ballverfolgung gelernt und wird nun an eine Wand gesetzt, zeigt er anfänglich das selbe Verhalten, d.h. er dreht sich zur Wand und läuft frontal gegen sie. Nachdem er mehrfach unter unterschiedlichen Winkeln auf die Wand zugefahren ist, ändert er sein Verhalten und lernt, vor der Wand abzudrehen. Das Umlernen funktioniert auch korrekt, wenn auch langsamer, wenn ein an der Steuerung beteiligter Sensor des Roboters ausfällt.

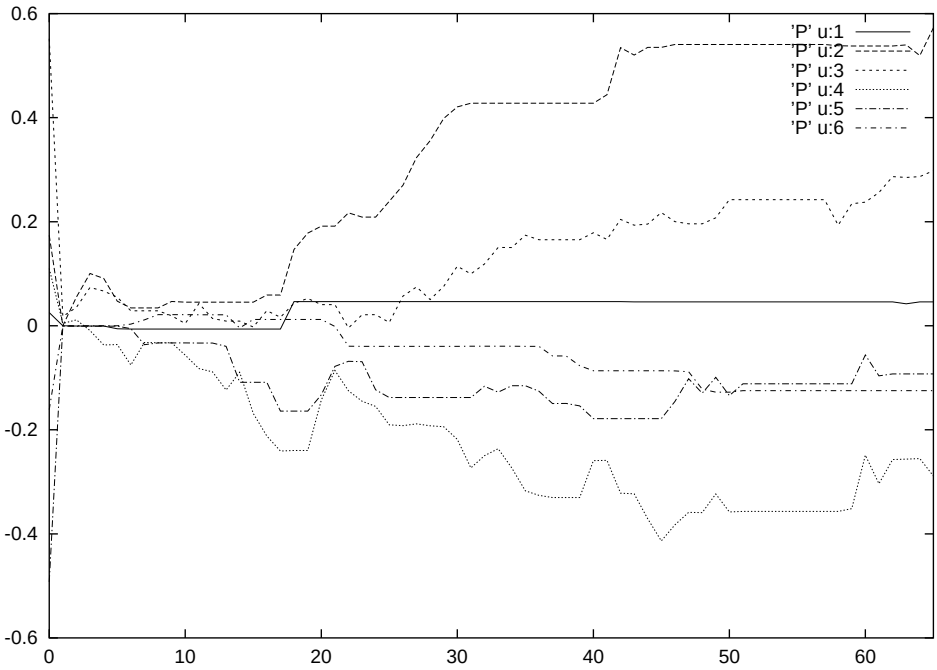


Abbildung 5.4: Ballverfolgung

5.7 Zusammenfassung

Die Experimente mit dem *khepera*-Roboter haben gezeigt, daß das Prinzip der *homeokinesis* ein sicheres Verfahren ist, um Verhaltensmuster autonomer Robotern zu erzeugen. In Abhängigkeit von den Umweltbedingungen wurde eine Vielzahl verschiedener Verhaltensmuster entwickelt, denen alle ein und dasselbe Controllermodell mit den gleichen initialen Controller- und Modellparametern und der gleichen Lernrate zugrunde lagen. Insbesondere wurden die folgenden Verhaltensmuster beobachtet:

- Kollisionsvermeidung
- Verfolgung bewegter Objekte
- Fahren durch Korridore
- verschiedene Modi der Wandverfolgung
- Bewegungsfähigkeit im komplexen Labyrinth, kein Festfahren

Desweiteren ist zu beobachten, daß die Parameter des Controllers zu einem Arbeitspunkt hin konvergieren, bei dem die Steuerung am besten ausgeführt werden kann.

Das Lernen erfolgt sehr schnell, d.h. es können hohe Lernraten ε gwählt werden ($\varepsilon = 0,01..0,3$). Das Lernen ist permanent aktiviert. In unveränderter Umgebung ist es reproduzierbar und konvergiert.

Zukünftige Arbeiten können mehrere nach dem Prinzip der *homeokinesis* arbeitende Module in einen einzigen größeren Controller integrieren. Die im Abschnitt 3.4.3 vorgestellte Hierarchie ist dazu geeignet. Ausgestattet mit einer noch zu formulierenden Bewertungsfunktion und der Fisher-Eigen-Dynamik kann der Metacontroller den Agenten auswählen, der sich am schnellsten anpassen bzw. lernen kann. Die Entwicklung sehr komplexer Verhaltensweisen wird damit möglich.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurden Grundbausteine für das Design eines Experimentiersystems für Lern- und Evolutionsexperimente mit autonomen Robotern, speziell mit dem *khepera*-Roboter, vorgestellt. Zuerst wurden die Anforderungen an ein solches System genau spezifiziert. Zur Realisierung des Systems wurden Methoden des objektorientierten Softwaredesigns eingesetzt. Controller-Funktionalitäten wurde in einzelne Module aufgeteilt und in einer Hierarchie angeordnet. Es wurden Verfahren zur

- Kollisionsdetektion,
- Fehlerbegrenzung und
- Erhöhung der Feinheit der Steuerung

vorgestellt. Desweiteren wurde eine effektive Kommunikationsschnittstelle entworfen. Das modulare Konzept des Controllers hat sich günstig für Erweiterungen erwiesen. Die konkrete Implementation hat sich bei der Entwicklung von zwei neuen Lernverfahren und bei mehreren Langzeitversuchen bewährt. Das vorgestellte System ist noch nicht komplett. Zukünftige Arbeiten können sich mit der Entwicklung von Controllermodulen für automatische Kollisionsauflösung und automatisches Homing befassen. Es können weitere Sensoren oder Informationsquellen (wie z.B. eine Overheadkamera zur Bahnverfolgung des Roboters) mit einbezogen werden. Die Software kann um Fernsteuerfunktionen erweitert werden, um von jedem Punkt im Internet einen Versuch überwachen zu können. Das Design einer Fitnessfunktion für Controllermodule ermöglicht auch die Konstruktion von hierarchisch geschachtelten Controllern oder einen vollautomatischen Betrieb der künstlichen *Roboterevolution*.

Anhang A

Das Programm *kcontrol*

A.1 Systemvorraussetzungen

Das Programm läßt sich auf jedem UNIX-Rechner mit grafischer Oberfläche (X11, X-Windows) und freiem Zugang zur seriellen Schnittstelle (Dateiberechtigungen) ausführen. Zum Kompilieren des Quellcodes sind neben den systemeigenen Entwicklungswerkzeugen und Header-Dateien die XForms-Bibliothek und die GNU-Werkzeuge *gcc, ld, sed, awk, make* und *install* notwendig. Vor dem Programmstart muß der Nutzer in seinem Heimatverzeichnis die Unterverzeichnisse

```
.kcontrol
.kcontrol/config
.kcontrol/data
.kcontrol/screenshots
.kcontrol/snns-nets
```

angelegt haben. Andernfalls können Zwischenergebnisse, Log- und Konfigurationsdaten nicht abgespeichert werden.

A.2 Kommandozeilenoptionen

Das Programm *kcontrol* wird in einem X-Terminal in einer Shell gestartet. Es benutzt zur Anzeige das XForms-Toolkit und die X11-Schnittstelle. Beim Start des Programms

können folgende Kommandozeilen-Optionen angegeben werden. Beim Fehlen dieser Optionen werden Standardwerte angenommen.

`-h, --host <Name>`	das Programm hat keinen Zugriff auf die serielle Schnittstelle und wird als Netzwerkclient gestartet. Das Argument bezeichnet den zu verwendenden Server. Die Voreinstellung ist *"localhost"*.
`-p, --port <Portnummer>`	das Programm hat keinen Zugriff auf die serielle Schnittstelle und wird als Netzwerkclient gestartet. Das Argument bezeichnet die zu verwendende Portnummer. Voreinstellung ist die vom Systemadministrator vorkonfigurierte Portnummer oder *55555*.
`-v, --visual <Nummer>`	gibt das bei Grafikanzeigen mit 8-bit Farbtiefe zu verwendende Farbpalettenmodell an, Voreinstellung ist *3*.
`-d, --device <Name>`	gibt den Pfad der seriellen Schnittstelle an, Voreinstellung ist */dev/ttyS*
`-i, --index <Nummer>`	gibt den Startindex der seriellen Schnittstelle bei der Verwendung mehrerer Schnittstellen an. Die Voreinstellung ist *0*.

A.3 Die Fenster

A.3.1 Hauptfenster

Kurz nach dem Programmstart erscheint auf dem Bildschirm das Hauptfenster. Durch Rahmen sind die folgenden Anzeigebereiche voneinander abgetrennt:

- Werte der Zeilenkamera (oben)
- Controller- / Algorithmenauswahl (mitte)
- Anzeigeparameter (mitte/links unten)
- Anzeige von Prozeß-Eigenschaften und Zeitverhalten (mitte unten)

- Anzeige der Umgebungslicht-, Enfernungs- u.a. Sensoren (rechts unten)
- Feld für interaktive Steuerung (links, ganz unten)

Im oberen Teil werden die Daten der Zeilenkamera dargestellt. Es existieren zwei Anzeigen für die Pixeldaten. Die erste zeigt die Bilddaten, so wie sie gelesen wurden. Im zweiten Anzeigefeld werden die gefilterten Pixeldaten gefiltert angezeigt. Als Filter stehen clipping, Hochpaß, Tiefpaß, Mittelung und Fouriertransformation zur Verfügung. Die Reihenfolge der Filterung kann mit Hilfe von Pull-Down-Menüs frei gewählt werden. Es können maximal 4 Filter in einer Kette hintereinander angeordnet werden. Rechts von diesen Anzeigen befindet sich die Darstellung der Blendenöffnung der Kamera.

Die mittlere Gruppe beinhaltet die Menüs zur Auswahl des aktiven Controllers, die Aktivierung der Algorithmen und deren Bildschirmanzeigemethoden sowie den Schalter zum Verlassen des Programms. Mit dem ersten Menü wird der aktive Controller festgelegt. Als aktiver Controller kann immer nur ein bestimmter Algorithmus ausgewählt werden. Mit dem zweiten Menü werden die Algorithmen ausgewählt, die während der Versuchsdurchführung ständig berechnet werden sollen. Algorithmen, die den Roboter nicht selbst steuern (z.B. ein Algorithmus zur Selbstlokalisation), können parallel mitaktiviert werden. Mit dem dritten Menü kann festgelegt werden, ob die berechneten Daten am Bildschirm angezeigt werden sollen.

Mit den Anzeigeparametern in der dritten Gruppe kann festgelegt werden, wieviel Sensordaten pro Zeitschritt vom Roboter zum Hostrechner übertragen werden und ob diese Daten am Bildschirm angezeigt werden sollen, da die Übertragung und Anzeige dieser Daten mitunter relativ viel Zeit in Anspruch nehmen kann. Weiterhin ist es möglich, die Auffrischrate für das Grafikdisplay zu verändern und den Inhalt des Grafikfensters in eine Bildersequenz zu sichern.

In der mittleren Gruppe werden die wichtigsten Eigenschaften zum Zeitverhalten des Programms ausgegeben. Das sind:

- die benötigte Zeit zum Senden eines Kommandos (in Millisekunden)
- die benötigte Zeit zum Empfangen eines Kommandos (in Millisekunden)
- die Dauer (Schrittweite) eines Intervalls (in Millisekunden)
- die Prozessorauslastung (in Prozent)

- die aufgewendete Zeit für das Datenloggen (in Millisekunden)
- die zum Synchronisieren von Roboter und Hostrechner aufgewendete Wartezeit (in Millisekunden)
- die Anzahl der Zeitschritte, bevor ein Kommando komplett abgearbeitet ist oder angeforderte Sensorinformationen vorliegen.

Mit den vorliegenden Daten kann entschieden werden, ob im Interesse einer genaueren Versuchsdurchführung auf die permanente Anzeige aller Sensorinformationen verzichtet wird. Das ist vor allem bei Hostrechnern mit geringerer Rechenleistung notwendig.

In der rechten Gruppe werden (sofern eingeschaltet) die aktuell verfügbaren Sensorinformationen des Roboters dargestellt. Es sind zwei mal acht senkrechte Balkenanzeigen für Licht- und Entfernungssensoren, eine waagerechte Balkenanzeige für die Breite der Greiferöffnung, eine Textausgabe, ein Lichtpunkt zur Anzeige der Objektpräsenz im Greifer, ein Fadenkreuz und zwei Winkelzeiger zur Positionsanzeige vorhanden.

In der letzten Gruppe links unten befinden sich die Druckknöpfe, um den Roboter zu drehen, vor und zurück zu fahren, den Greifarm zu bedienen und den ausgewählten Versuch zu starten. Weiterhin kann mit einem Schieberegler die erlaubte Höchstgeschwindigkeit des Roboters bestimmt werden.

A.3.2 Parameter-Fenster

Wird vom Nutzer ein Algorithmus ausgewählt, öffnet sich ein weiteres Fenster auf dem Bildschirm mit Schieberegler- und Druckknopf-Elementen. Jeder Schieberegler repräsentiert den Wert einer Variablen im jeweiligen Algorithmus. Die Druckknöpfe repräsentieren Boolesche Variablen im Algorithmus. Ihr Zustand ist "*wahr*", wenn der jeweilige Knopf gedrückt ist.

Jeder Algorithmus hat mindestens 2 Schieberegler und 3 Druckknöpfe für die Parameter, die in allen Algorithmen genutzt werden. Alle weiteren Elemente sind algorithmenspezifisch.

A.3.3 Schieber-Grenzen

Durch Drücken der rechten Maustaste auf einen Schieberknopf aktiviert man ein kleines Menü für die Bereichsgrenzen des jeweiligen Schiebers. In den Feldern der ersten

Typ	Bezeichnung	Erklärung
Schieber	log.intvl.	Intervall, in dem Logdaten geschrieben werden
Schieber	log.avg.	Anzahl der Zeitschritte, über die die Logdaten gemittelt werden
Checkbox	load on start	beim Start wird die Konfigurationsdatei für diesen Algorithmus gelesen
Checkbox	save on stop	bei Stopp wird die Konfigurationsdatei für diesen Algorithmus geschrieben, alle Schiebereinstellungen werden gesichert
Checkbox	reset	die Rücksetz-Methode des Algorithmus-Objektes wird beim Drücken einmal aufgerufen

Tabelle A.1: generische Bedienelemente für alle Demos

Zeile kann die obere und untere Grenze angegeben werden. In der zweiten Zeile befinden sich die Eingabefelder für die Anzeigegenauigkeit (Stellen hinter dem Komma) und die Schrittweite. Mit der Schrittweite wird die Mindeständerung vorgegeben, die bei einer Bewegung des Schiebers erfolgen soll. Die Daten werden aus einem Feld erst übernommen, wenn der Fokus in ein anderes Feld gewechselt hat. Mit dem Druckknopf "Cancel" auf der linken Seite können die Änderungen rückgängig gemacht werden. Das Menü wird beendet durch Drücken des "OK"-Knopfs. Dabei werden die neuen Einstellungen übernommen, und der Schieber kann in den neuen Grenzen bewegt werden.

A.3.4 Grafik-Ausgabe

Sofern ein Algorithmen-Objekt über eine implementierte "Show"-Methode verfügt und dieser Algorithmus zur Anzeige ausgewählt wurde, öffnet sich das Grafik-Fenster (Grafik-Canvas). Der Inhalt des Fensters kann durch Drücken der Taste "F2" in eine Datei gesichert werden. Das Sichern erfolgt aus Platzgründen im RLE-Format. Bevor diese Dateien betrachtet werden können, müssen sie mit dem Filter `rle2ppm` in das PPM-Format umgewandelt werden.

A.4 Einkompilierte Algorithmen/Demos

A.4.1 Start/Stopp von einkompilierten Algorithmen

Ein Demo wird gestartet, indem auf den Button *"Run"* im Feld für die interaktive Steuerung geklickt wird. Die LED an diesem Button leuchtet gelb, wenn ein Demo

ausgeführt wird. Über die Popup-Menüs *"Controller"*, *"Algorithm"* und *"View"* lassen sich die zur Verfügung stehenden Controller-Agenten (siehe 3.1) aktivieren. Diese Auswahl muß vor dem Start geschehen. Jeder aktivierte Controller-Agent führt als erstes seine Startmethode aus. Wenn das Kontrollkästchen *"load on start"* des jeweiligen Parameter-Fensters aktiviert wurde, werden alle Variablen mit den gesicherten Parametern des letzten Durchlaufs initialisiert.

Jeder Controller-Agent sichert während eines Laufes kontinuierlich seine Logdaten in eine eigene Datei. Der Umfang läßt sich durch die Schieberegler *"log.intvl"* und *"log.avg"* beeinflussen. Das Log-Intervall ist die Anzahl von Zeitschritten zwischen dem Sichern von zwei aufeinanderfolgenden Log-Datensätzen. Ist dieser Wert Null, wird nicht geloggt. Der Wert *"log.avg"* bestimmt das Zeitfenster in dem Log-Daten gemittelt werden, um irreführende Treppeneffekte in den Logdaten zu vermeiden. Dieser Wert gilt nur, wenn das Logging-Intervall größer als ein Zeitschritt ist. Die Formel, mit der die Logdaten gemittelt werden, ist:

$$\begin{aligned} LogData_{neu} &= LogData_{alt} + \Delta LogData \\ \Delta LogData &= \frac{(Data - LogData_{alt})}{loadavg} \end{aligned}$$

Das Anhalten erfolgt durch erneutes Klicken auf den *"Run"*-Button. Jeder aktivierte Controller-Agent führt dabei als letztes seine Stoppmethode aus. Diese Methode sichert den aktuellen Parametersatz, wenn das Kästchen *"save on stop"* aktiviert war.

A.4.2 Neuronalcontroller

Der Neuronal-Controller besitzt keine Berechnungsmethode. Er läßt sich damit mit allen anderen Controlleragenten kombinieren, die den Roboter nicht selbst steuern müssen. Aktiviert wird der Controller durch Anwählen im Controller-Popup-Menu. Danach öffnet sich ein Dialog zum Laden der Netzwerk-Datei. Diese Datei muß im SNNS-Format[1] erstellt worden sein und kann ein einfaches Feed-Forward- oder rekurrentes Netz enthalten. Das Netz kann maximal soviel Eingänge (Ausgänge) haben wie der Roboter Sensoren (Aktuatoren) hat. Die Abbildung, welcher Sensor (Aktuator) welchen Neuronen zugeordet ist, erfolgt in der Quelltextdatei `net2robot.h`. Die Skalierung der Ein-/Ausgabewerte auf die Anforderungen des Roboters erfolgt durch das dem einfachen Neuron überlagerte Objekt `t_IO_Neuron`.

[1] Stuttgart-Neuronal-Network-Simulator

Dieses Objekt besitzt keine Logging-Methoden. Als einziger Parameter kann nur die Skalierung für die Geschwindigkeit des Roboters eingestellt werden. Im Grafik-Canvas wird nur die Topologie des Netzwerkes angezeigt.

A.4.3 Lichtkompaß

Der Lichtkompaß ist ein Controller ohne eigene Ausführungsmethode. Die Werte der Umgebungslichtsensoren werden durch eine Kreuzkorrelation mit einer Gauss-Kurve auf einen Winkel abgebildet. Aus diesem Winkel "sieht" der Roboter eine Lichtquelle. Der Lichtkompaß hat Methoden, um auf dem Grafik-Canvas zu zeichnen. Es werden die Sensorwerte (als Balkengrafik) und die aktuelle Ausrichtung zur Lichtquelle (als Zeiger) dargestellt. Die von der Odometrie abgeleitete Winkelangabe wird mit dem grauen Zeiger dargestellt.

A.4.4 Pfadsimulation

Die Pfadsimulation besitzt nur eine Ausführungsmethode und dient zur Wiederholung eines Laufes. Dieser Controller kann ebenfalls nur mit Controller-Agenten, die den Roboter nicht selbst steuern, angewendet werden. Nach der Auswahl dieses Kontrollers wird in einem Dialog zur Eingabe des Dateinamens der Trajektorien-Datei aufgefordert. Diese Datei ist typischerweise eine Sequenz von x/y-Koordinaten, die bei einem vorhergehenden Lauf des Roboters mit Hilfe der Odometrie aufgezeichnet worden sind. Die Aufzeichnung des Weges ist grundsätzlich aktiviert.

Mit den Schiebern im Parameter-Fenster kann die Ausführungsgeschwindigkeit beschleunigt oder verzögert und die maximale Pfadlänge in der Anzeige auf dem Grafik-Canvas begrenzt werden.

A.4.5 Wettbewerb der Differentialgleichungen

Diesem Demo liegt der in Kapitel 4 beschriebene Algorithmus zugrunde. Es wurden 4 Geradengleichungen implementiert. Bei aktivierter Anzeigemethode werden diese Geraden als Strahlen im Grafik-Canvas angezeigt. Jeder Strahl ist mit seiner Nummer, Wahrscheinlichkeit aus Gleichung 4.10 und seinem Fehler (Gleichung 4.5) beschriftet. Die Gerade mit dem geringsten Fehler ist mit einen blauen Kreis am Ende markiert. Die Gerade, die den Update gewonnen hat, ist gelb gezeichnet, alle anderen sind rot

gezeichnet. Zur Visualisierung der Verdrehungskorrektur nach Gleichung 4.22 wird ein grüner Zeiger gezeichnet. An dessen Ende ist die aktuelle Verdrehungsgeschwindigkeit nach Gleichung 4.21 beschriftet. Im Verlauf des Versuchs werden die Geraden bewegt und zeigen in die Richtung der vier am häufigsten gegangenen Wege. Das ist gut zu erkennen, wenn die Anzeigemethode der Pfadsimulation ebenfalls aktiviert ist.

Da in diesem Demo nur die Berechnungs- und die Anzeigemethode implementiert sind, wird noch ein weiterer Controlleragent, der den Roboter steuert, benötigt. Das können ein simpler Wandfolgealgorithmus, ein auf Wandverfolgung trainiertes neuronales Netz oder die Daten aus einem früheren Lauf (Simulation) sein. Mit den Schiebereglern lassen sich die Parameter Lernrate (ε aus Gleichung 4.9), Umschaltgeschwindigkeit (η aus Gleichung 4.8) und die Updateverzögerung (Kapitel 4, Punkt 4.4.1.3) in Zeitschritten während des Versuchs verändern.

A.4.6 *homeokinesis*-Controller

Diesem Demo liegt der in Kapitel 5 beschriebene Algorithmus zugrunde. Es gibt eine Steuerungs- und eine Berechnungsmethode. Die Anzeigemethode ist leer, d.h. es wird nichts angezeigt. Die Bedienelemente steuern die folgenden Variablen des Algorithmus.

A.4.6.1 Schieber

frequenz Periode der Störung, Variable T aus Gleichung 3.4. Für die Funktion $\phi(t)$ wurde der *Sinus* gewählt.

lernrate Lernrate für die Controllerparameter c_i, entspricht ε aus Gleichung 5.17

horizont zeitlicher Abstand von zwei Sensorwerten, die zur Differenzbildung benutzt werden, sinnvolle Werte sind 1 und 2 (ganzzahlig).

störung Verstärkungsfaktor der Störung, ε aus Gleichung 3.4, sinnvolle Werte sind $0,05 .. 0,2$.

speed Robotergrundgeschwindigkeit, Standardwert ist $1,0$.

amplify Faktor für maximale Kurvengeschwindigkeit. Für Werte $> 1,0$ kann das kurveninnere Rad auch rückwärts drehen, für Werte $< 1,0$ bleibt das kurveninnere Rad in keinem Fall stehen, siehe auch Definition der Krümmung in Kapitel 3.6.2.

A.4.6.2 Checkboxen

add noise Störung/Funktion $\phi(t)$ einschalten

init random/load wenn gedrückt, zufällige Initialisierung der Controllerparameter c_i beim Start, ansonsten Laden der Parameter vom vorherigen Versuch

wakeup gelernt wird nur, wenn für die Sensorwerte $x_i < 1,0$ gilt.

A.5 Verschiedenes

A.5.1 Softwareseitige Erweiterungen

Für Erweiterungen um neue Controllermodule steht das Skript mk_newdemo zur Verfügung. Es wird mit

```
mk_newdemo <Name> [ENTER]
```

aufgerufen. Es kopiert die Templates für eine C++- und eine Headerdatei nach *<Name>*`_demo.cc` und *<Name>*`_demo.h`. Weiterhin löscht es die Datei `.sources` im aktuellen Verzeichnis und stellt damit sicher, daß die neuen Dateien beim nächsten Aufruf von `make` berücksichtigt werden. In den neuen Dateien können mit einem beliebigen Editor die leeren Funktionsrümpfe mit dem gewünschten Code gefüllt werden. Damit das neue Controllerobjekt im Programm bekannt wird, muß noch in der Datei `user.cc` die Kompilerdirektive
`#include "<Name>_demo.h"` aufgenommen und in der Funktion `user_cb_init()` die Instanz des neuen Controllers an die Kette der anderen Controller angehängt werden. Danach kann `make` aufgerufen werden. Der neue Controller sollte nun, entsprechend seiner Flags, in den Pulldown-Menüs für Steuerung, Berechnung und Grafikanzeige zu sehen sein.

A.5.2 Trajektorienaufzeichnung

Die Trajektorie des Roboters während eines Versuchs wird immer in die Datei `.kcontrol/data/log_SimPath.txt` aufgezeichnet. Wenn ein neuer Versuch gestartet

wird, wird diese Datei mit der neuen Trajektorie überschrieben. Soll eine bestimmte Trajektorie gesichert werden, muß zwischen zwei Versuchen diese Datei umbenannt werden.

Das Loggen von Sensorwerttrajektorien kann einfach über einen neuen Controlleragenten implementiert werden, bei dem nur die Methoden `make_logstring()` und `make_logdata()` überlagert werden.

A.5.3 Bibliotheken

Die Software wurde mit Hilfe von dynamisch linkbaren Bibliotheken realisiert.

1. libkhepera.so

   In dieser Bibliothek befinden sich die Funktionsprimitive des *khepera*-Roboters.

2. libtns_util.so

   Diese Bibliothek enthält generische Funktionen zur Zeitmessung, TCP-Socket Ein-/Ausgabe, einen Sortieralgorithmus (Quicksort) und weitere kleine Funktionen und Makros, die das Programmiren effizienter machen.

3. libxfgc.so

   Diese Bibliothek enthält Routinen zur Grafikanzeige in Verbindung mit der Toolkit-Bibliothek.

4. libforms.so

   Toolkit-Bibliothek für das grafische Benutzerinterface auf X11-Displays.

Die Bibliothek libforms.so ist Bestandteil des XForms-Toolkits von T. C. Zhao und Mark Overmars. Die anderen drei Bibliotheken wurden vom Autor entwickelt.

Anhang B

Objektreferenz

B.1 Klassen des Hauptprogramms

class *<classname>*

Datei: `<wo_definiert>`

Zweck: Kurzbeschreibung, alle Objekte besitzen eine *Init()*- und eine *Done()*-Methode, diese Methoden werden nur im weiteren Text erwähnt, wenn sie sich von den folgenden Deklarationen unterscheiden

externe Methoden:

`void Init(void);`
initialisiert das Objekt

`void Done(void);`
bereinigt das Objekt

class *t_avg_group*

Datei: `CrashTest_demo.h`

Zweck: Hilfsobjekt, berechnet mehrere Gruppen von Durchschnittswerten

externe Methoden:

`void Init(void);`
initialisiert das Objekt

`void Add(double s, double g, int phase);`
läßt die Werte *s, g* und *phase* in die Durchschnittsberechnung einfließen

class *t_special_counter*

Datei: `t_special_counter.h`

Zweck: Hilfsobjekt

externe Methoden:

`void Init(unsigned long d, int f, int l);`
: initialisiert das Objekt

`char Start(char bools);`
: startet den Zähler, wenn die Variable *bools* wahr ist

`void Stop(void);`
: stoppt den Zähler

`void Count(char bools);`
: inkrementiert den Zähler, wenn die Variable *bools* wahr ist

class *t_trigger*

Datei: `t_trigger.h`

Zweck: Hilfsobjekt zum Auslösen von Ereignissen

externe Methoden:

`void Init(int len);`
: initialisiert das Objekt mit der Anzahl der zu verwaltenden Ereignisse (*len*)

`void Register(int id, long int ms);`
: meldet ein Ereignis mit Kenn-Nummer *id* und Timeout *ms* an

`void Update(void);`
: Callback-Routine, muß einmal pro Arbeitsschritt aufgerufen werden

`char Snapped(int id);`
: gibt den Wert *wahr* zurück, wenn für das betreffende Ereigniss eine Zeitüberschreitung registriert wurde

class *t_Neuron*

Datei: `neuronal.h`

Zweck: Basisobjekt, Modell eines Neurons für Feed-Forward und rekurrente neuronale Netze, kann in eine verkettete Liste aufgenommen werden

externe Methoden:

`void Init(int i, t_vector *p, int t, double b);`

initialisiert ein Neuron vom Typ t mit Identifikationsnummer i und Bias b an Position p

`char InBox(int x, int y);`

gibt den Wert *wahr* zurück, wenn die Koordinaten x und y nahe an der eigenen Position sind. Diese Methode wird nur für Bildschirmausgaben gebraucht.

`class t_Neuron *GetNeuron(int id);`

gibt einen Zeiger auf ein Neuron mit der Identifikationsnummer *id* zurück, diese Methode sucht in der verketteten Liste

`void Cut(class t_Neuron *);`

schneidet ein Neuron aus der verketteten Liste heraus. Diese Methode sucht in der verketteten Liste.

`void Cut(class t_Synapse *);`

schneidet eine Verbindung zwischen zwei Neuronen heraus

`void Connect(class t_Neuron *other, double w);`

verbindet das Neuron mit dem im Argument *other* übergebebenen Neuron durch eine Synapse mit dem Gewicht w (siehe auch `t_synapse`)

`char IsActive(void);`

gibt den Wert *wahr* zurück, wenn der Wert der Aktivierung des Neurons größer als 0.5 ist

`virtual double Activation(int stamp);`

gibt den Aktivierungszustand eines Neurons aus. Die Zeitmarke *stamp* wird in rekurrenten Netzwerken benutzt. Eingabe-Neuronen überlagern diese Methode, um die Eingabedaten dem Netz zuzuführen.

`virtual void WriteOutput(int stamp);`

nur Funktionsprototyp, muß überlagert werden. Ausgabe-Neuronen können diese Methode aufrufen, um eine spezielle Ausgabeaktion durchzuführen.

`virtual void Show(void);`

nur Funktionsprototyp, muß überlagert werden und soll das Objekt am Bildschirm anzeigen.

class *t_Synapse*

Datei: `neuronal.h`

Zweck: Modell einer Synapse, kann in eine verkettete Liste aufgenommen werden

externe Methoden:

`void Init(int id);`
: initialisiert das Objekt mit der Identifikationsnummer *id*

`double ReadInput(int);`
: summiert über alle Ausgaben der verbundenen Neuronen und gibt diesen Wert zurück

`void Cut(class t_Neuron *);`
: trennt die Verbindung zu einem Neuron. Diese Methode sucht in der verketteten Liste.

`virtual void Show(t_vector &);`
: nur Funktionsprototyp, muß überlagert werden und soll das Objekt am Bildschirm anzeigen.

class *t_IO_Neuron : public t_Neuron*

Datei: `net2robot.h`

Zweck: abgeleitetes Objekt, realisiert spezielles Ein-/Ausgabe-Neuron

externe Methoden:

`void Init(RobotContext *R, int i, t_vector *p, int t, double b, double *s);`
: initialisiert ein Ein-/Ausgabeneuron wie in `t_Neuron::Init(...)` und stellt die Verbindungung zu den Sensoren und Aktuatoren des Roboters her

`virtual void WriteOutput(int stamp);`
: Methode zur Bedienung der Aktuatoren

`virtual double Activation(int stamp);`
: Methode zum Lesen der Sensordaten

class *t_RobotNet*

Datei: `net2robot.h`

Zweck: Modell eines neuronalen Netzes mit Verbindungen zu den Sensoren und Aktuatoren eines Roboters

externe Methoden:

`virtual void Init(RobotContext *R, double *s);`

initialisiert das Netzwerk mit Verbindung zum Roboter R und Zeiger auf die Geschwindigkeitsvariable s

`void Load(const char *fn);`

lädt ein in der Datei *fn* definiertes neuronales Netz

`t_Neuron *GetNeuron(int x, int y);`

gibt einen Zeiger auf eine Instanz eines Neurons zurück, das nahe an der Position *(x,y)* liegt

`void CutSynapse(t_Neuron *, t_Neuron *);`

schneidet eine Verbindung zwischen zwei Neuronen heraus

`void CutNeuron(t_Neuron *);`

schneidet ein Neuron aus dem Netz heraus

`virtual void Show(void);`

überlagerte Methode, zeichnet das Netz

`void Processing(void);`

führt einen Arbeitsschritt im Netz aus

class *t_SOM*

Datei: `neuro_gas.h`

Zweck: Modell einer selbstorganisierenden Karte

externe Methoden:

`void Init(int w, int h);`

initialisiert die Karte mit $w * h$ Neuronen

`void Create(int w, int h);`

erzeugt eine Karte mit Breite w und Höhe h

`t_Neuron *GetWinner(t_vector *p, int c);`

ermittelt das am nächsten zur Position p liegende Neuron und gibt einen Zeiger darauf zurück. Wenn der Wert c wahr ist, wird dabei eine Neuberechnung durchgeführt.

`t_Neuron *Add(t_vector *p);`

fügt der Karte ein Neuron an Position p hinzu

`char Update(t_vector *p);`

führt einen Update-Schritt zur Position p für die gesamte Karte aus

`void ProcessInput(t_vector *, int);`

Callback-Routine für einen Zeitschritt

`void ResetLinks(t_Neuron *);`

setzt die Gewichte aller Verbindungen auf den Wert *0* zurück

`void CalcSize(void);`

berechnet die Ausdehnung des Netzes in x- und y-Richtung neu

`void Show(void);`

zeichnet die Karte auf den Bildschirm

class *t_runtime_object*

Datei: `t_runtime_object.h`

Zweck: Modell eines Laufzeitprozesses, enthält Methoden und Funktionsprototypen zur Bearbeitung von in jedem Demo bzw. Testalgorithmus gleichermaßen vorkommenden Aufgaben

externe Methoden:

`virtual char calculate_module(char lerne, char neu);`

nur Funktionsprototyp, muß überlagert werden und soll Berechnungsroutinen aufrufen.

`virtual char control_module(void);`

nur Funktionsprototyp, muß überlagert werden und soll die Steuer- und Ausgaberoutinen aufrufen.

`virtual char start_module(void);`

startet (aktiviert) die Abarbeitung dieses Moduls

`virtual char stop_module(void);`

hält die Abarbeitung dieses Moduls an

`virtual void pause_module(char);`

versetzt ein Modul in den Zustand *Pause*

`virtual void make_logdata(void);`

nur Funktionsprototyp, muß überlagert werden, soll Logdaten aufbereiten und wird in jedem Arbeitsschritt aufgerufen.

`virtual void make_logstring(void);`

nur Funktionsprototyp, muß überlagert werden, soll Logdaten aufbereiten und wird nur beim Schreiben von Logdaten aufgerufen.

`virtual void Show(void);`

ruft alle Anzeigemethoden von angeschlossenen Objekten auf

`virtual void Reset(void);`

setzt alle internen Variablen auf den Startzustand zurück, kann mehrfach überladen werden

`virtual void Save(int fd);`

sichert die aktuellen Einstellungen in die Datei mit dem Dateideskriptor *fd*

`virtual void Load(char *to_parse);`

sucht in einer Zeichenkette Konfigurationsoptionen, wird für jede Zeile einer Konfigurationsdatei aufgerufen, kann mehrfach überladen sein

`virtual void button_cb(FL_OBJECT *ob,long data);`

nur Funktionsprototyp, muß überlagert werden, soll die notwendigen Aktionen auf Nutzereingaben (Tastendrücke) ausführen

`virtual void slider_cb(FL_OBJECT *ob,long data);`

nur Funktionsprototyp, muß überlagert werden, soll die notwendigen Aktionen auf Nutzereingaben (Schieberegler) ausführen

`virtual void Interactions(char show);`

öffnet die Dialogfenster des Prozesses

class *t_runtime_root*

Datei: `t_runtime_object.h`

Zweck: Verwaltung von mehreren Laufzeitprozeß-Objekten vom Typ `t_runtime_object`

externe Methoden:

`void Add(t_runtime_object *o);`

meldet den Laufzeitprozess *o* an

`void Remove(t_runtime_object *o);`

meldet den Laufzeitprozess *o* ab

`void Execute_Calc(char do, char lerne, char neu);`

führt alle Berechnungsroutinen von angemeldeten und aktiven Prozessen aus. Wenn die Variable *lerne* den Wert *falsch* hat, wird das Lernen von enthaltenen Lernprozessen nicht erlaubt. Die Variable neu (Wert *falsch*

oder *wahr*) zeigt das Vorhandensein von neuen/geänderten Datensätzen an.

`void Execute_Cntl(char do, char pause);`
führt alle Steuer- und Ausgaberoutinen von angemeldeten und aktiven Prozessen aus

`void Show(void);`
ruft alle Anzeigemethoden von angemeldeten und aktiven Prozessen auf

`void do_slider(FL_OBJECT *ob, long data);`
ruft alle Schieberegler-Interaktionsroutinen von angemeldeten und aktiven Prozessen auf

`void do_button(FL_OBJECT *ob, long data);`
ruft alle Tastendruck-Interaktionsroutinen von angemeldeten und aktiven Prozessen auf

`char **find_slidername(FL_OBJECT *ob, int &ron);`
findet zu einem XForms-Anzeige-Objekt den Index *ron* in der Tabelle der Laufzeitprozesse und gibt einen Zeiger auf den Namen des XForm-Objektes zurück

`char NeedCanvas(void);`
prüft alle angemeldeten und aktiven Prozesse und gibt den Wert *wahr* zurück, wenn das Freiform-Graphik-Fenster zur Anzeige benötigt wird.

class *t_deq*

Datei: `t_deq.h`

Zweck: Basisobjekt, Modell einer Differentialgleichung, kann in eine verkettete Liste eingehängt werden, wird im *diffeq*-Demo verwendet

externe Methoden:

`void Init(int i, double a, t_deq *N);`
initialisiert das Objekt mit Identifikationsnummer i, Startparameter a und nachfolgendem Listenelement N. Das Argument N muß beim letzten Listenelement *NULL* sein.

`virtual void Show(t_vertex_2i *C, double S, t_deq *P, t_deq *F, t_deq *E);`
nur Funktionsprototyp, muß überlagert werden und soll das Objekt auf dem Bildschirm anzeigen

`virtual double Error(t_vector &Df, double ds, double dav, double dsv);`

nur Funktionsprototyp, muß überlagert werden und soll den aktuellen Fehler berechnen und zurückgeben

`virtual void Update(double, t_vector &, double, double, double);`

leerer Funktionsrumpf, muß von abgeleiteten Objekten überlagert werden

`double Fit(double err_sum);`

gibt die aktuelle Fitness des Objektes bezogen auf den Gesamtfehler *err_sum* aus

`double angle_diff(double tan_a2);`

interpretiert die internen Parameter eines Differentialgleichungsobjektes als Tangens eines Winkels und gibt die Winkeldifferenz zu dem zu *tan_a2* gehörenden Winkel aus.

`char ComparesTo(t_deq *other);`

vergleicht zwei generische Differentialgleichungsobjekte, gibt den Wert *wahr* zurück, wenn die Summe über alle Absolutwerte der Differenzen der Parameter kleiner als 0.01 sind.

class *t_deq_l : public t_deq*

Datei: `t_deq_l.h`

Zweck: von t_deq abgeleitetes Objekt speziell für Geraden

externe Methoden:

`virtual void Show(...);`

überlagerte Methode, gibt das Objekt am Bildschirm aus

`virtual double Error(...);`

überlagerte Methode, berechnet den aktuellen Fehler

`virtual void Update(...);`

überlagerte Methode, führt ein Update auf das Objekt aus

class *t_deq_c : public t_deq*

Datei: `t_deq_c.h`

Zweck: von t_deq abgeleitetes Objekt speziell für Kreisbahnen

externe Methoden:

`virtual void Show(...);`

überlagerte Methode, gibt das Objekt am Bildschirm aus

`virtual double Error(...);`

überlagerte Methode, berechnet den aktuellen Fehler

`virtual void Update(...);`

überlagerte Methode, führt ein Update auf das Objekt aus

class *t_parameter*

Datei: `t_polynom.h`

Zweck: Objekt für Exponentialkoeffizienten von Polynomen, findet im Objekt `t_Polynom` Verwendung und vereinfacht Logging und Batchmodus

externe Methoden:

`void Init(double l, int n);`

initialisiert das Objekt mit dem Mittelungsinterval l für Log-Daten und der Schrittweite n für den Batchmodus. Der Parameter n muß größer bzw. gleich eins sein.

`double Get(void);`

gibt den gespeicherten Wert zurück

`double Real(void);`

gibt den aktuellen Wert zurück

`void Set(double d);`

setzt die Variablen für den gespeicherten, den aktuellen und den Datenlogger-Wert auf den Wert d

`void Assign(void);`

wird in jedem Arbeitsschritt einmal aufgerufen, übernimmt jeweils nach n Arbeitsschritten den internen Wert in den gespeicherten Wert

`void AddV(double d);`

addiert d zum gespeicherten Wert

`void Reset(void);`

macht den letzten Aufruf von *AddV()* rückgängig

`void AddS(double d);`

addiert d zum internen Wert

class *t_Polynom*

Datei: `t_polynom.h`

Zweck: Modell eines Polynoms der Form $P = \sum_{n=0}^{N} c_{i,n} x_i^n \quad i\epsilon\{0,..,7\}$, wird im *homeokinesis*-Demo verwendet

externe Methoden:

`void Grad(double x[], double h, double e, double d, char w);`

berechnet den Gradienten des Gesamtpolynoms an der Stelle x in der Umgebung h

`double Result(double x[]);`

berechnet das Gesamtpolynom an der Stelle x

`void Init(int g, int n);`

initialisiert das Gesamtpolynom als eine Gruppe von n Teilpolynomen vom Grad g

`void Init(t_Polynom *P);`

initialisiert eine Polynomgruppe als Kopie einer anderen

`void Reset(t_Polynom *P);`

übernimmt alle Einstellungen und Werte von einem anderen Polynom

`int Index(int i, int n);`

macht den zweidimensionalen Index "i-te Komponente vom n-ten Polynom" eindimensional

`void SetVal(int i, int j, double d);`

setzt den Koeffizient $c_{i,j}$ auf den Wert d

`void SetVal(int idx, double d);`

wie vorige Funktion, aber mit eindimensionalem Index

`void UseVal(int i, int n, char d);`

Die i-te Komponente des n-ten Teilpolynoms wird bei der Berechnung des Gesamtpolynoms nicht benutzt, wenn die Variable d den Wert *falsch* hat.

`void UseVal(int idx, char d);`

wie vorige Funktion, aber mit eindimensionalem Index

`double AddVal(int i, int n, double d);`

addiert den Wert d zum i-ten Koeffizienten des n-ten Teilpolynoms

`double AddVal(int idx, double d);`

wie vorige Funktion, aber mit eindimensionalem Index

`void BatchParm(int s);`

setzt die Schrittweite s im Batch-Modus, s muß größer als 0 sein

`double C_Val(int i, int n);`

gibt den i-ten Koeffizienten des n-ten Teilpolynoms zurück

`char C_used(int i, int n);`

gibt den Wert *wahr* zurück, wenn die i-te Komponente des n-ten Polynoms bei der Berechnung des Gesamtpolynoms mit berücksichtigt wird

`double C_logV(int i, int j);`

gibt den für das Logging gemittelten Wert des i-ten Koeffizienten vom n-ten Teilpolynom zurück

`void GetMem(void);`

interne Funktion zur Speicherbereitstellung

B.2 Klassen der Roboter-Bibliothek *libkhepera*

class *t_world*

Datei: `world.h`

Zweck: Objekt für die Bereichsgrenzen der virtuellen Welt des Roboters in x- und y-Richtung. Die Vektoren min(x1,y1) und max(x2,y2) spannen ein Rechteck auf.

externe Methoden:

`void Update(t_vector &v);`

Hier wird geprüft, ob der übergebene Punkt v innerhalb des von *min* und *max* aufgespannten Rechtecks liegt. Ist das nicht der Fall, werden die Vektoren so vergrößert (negative werden verkleinert), daß die obengenannte Bedingung wieder erfüllt ist. Dabei wird die interne Variable *grow* auf den Wert *true* gesetzt. Zurückgesetzt wird sie erst, wenn der übergebene Punkt v von vornherein innerhalb des Rechtecks liegt.

`void WrapMove(t_vector &v);`

Wenn der Vektor v außerhalb des bekannten Bereiches liegt, werden die Grenzen so verschoben, daß der Vektor wieder innerhalb des bekannten Bereichs liegt. Bildlich gesprochen verschiebt der Roboter einen übergestülpten Pappkarton (die Welt), wenn er von innen gegen eine

Wand des Pappkartons fährt.

`void Draw(void)`

zeichnet ein Viereck formatfüllend in ein Grafikfenster (Canvas)

class *t_PATH*

Datei: `t_path.h`

Zweck: Objekt für Trajektorien

externe Methoden:

`void Init(t_vector *center);`

initialisiert eine Trajektorie mit ihrem Startpunkt

`t_PATH *Add(t_vector *np);`

fügt einen neuen Punkt der Trajektorie hinzu und übergibt den neuen Anker auf die Trajektorie

`void Draw(t_world *W);`

zeichnet die komplette Trajektorie in den Grenzen der Welt W vom Ursprungspunkt an

`void Draw(t_world *W, int n);`

zeichnet die Trajektorie in den Grenzen der Welt W mit einem Startpunkt, der höchstens n Zeitschritte zurückliegt

`t_vector *LastPos(void);`

gibt den letzten Punkt (das Ende) der Trajektorie als Vektor zurück

`t_PATH *LastPos(int n);`

gibt den n -ten Punkt vor dem Ende der Trajektorie zurück

`t_PATH *Last_not_Adjusted(void);`

gibt bei teilweise korrigierten Trajektorien den letzten, noch nicht korrigierten Punkt zurück

`t_PATH *NextPos(FILE *fs);`

fügt der Trajektorie einen neuen Punkt hinzu. Die Koordinaten werden aus einer Textdatei gelesen

`int Length(void);`

gibt die Gesamtlänge der Trajektorie in Schritten zurück

`int Length(t_PATH *);`

gibt die Länge der Trajektorie in Schritten bis zu einem bestimmten Punkt

zurück

`void Save(char *fn);`

sichert die Trajektorie in eine Datei. Das Argument ist der Name der Datei.

`void Save(int fd);`

Sichert die Trajektorie in eine Datei. Das Argument ist der Filedeskriptor einer schon geöffneten Datei.

class *t_Range*

Datei: `t_floating_avg.h`

Zweck: Objekt zur Normierung von beliebigen, zum Programmstartzeitpunkt unbekannten Bereichsgrenzen

externe Methoden:

`char Update(double d)`

gibt den Wert *false* zurück, wenn *d* innerhalb der bekannten Grenzen liegt. Sonst wird der Wert *true* zurückgegeben und die Grenzen werden so verschoben, daß eine Normierung zwischen 0 und 1 erhalten bleibt.

`void Reset(void);`

setzt das Objekt zurück

class *t_laverage : public t_Range*

Datei: `t_floating_avg.h`

Zweck: berechnet gleitendes Mittel

externe Methoden:

`void Add(double v)`

fügt den nächsten Wert hinzu

`double Range(void);`

gibt die Differenz zwischen dem bisher größten und kleinsten Wert zurück

class *t_Sensor*

Datei: `t_sensor.h`

Zweck: Objekt für selbstkalibrierenden (selbstnormierenden) Sensor

externe Methoden:

`void Init(double loc, double avg, int len);`

initialisiert das Objekt mit einem Erwartungswert *avg* für den Durchschnitt und der Länge *len* des gewünschten Fensters für das gleitende Mittel. Der optionale Wert *loc* ist die Ausrichtung (Blickrichtung) des Sensors zum Roboter in Grad.

`void Set(double d);`

Der real gemessene Sensorwert *d* wird in das Objekt eingespeist. Alle internen Variablen werden abgeglichen. Die Variable *Value* ist der auf den Bereich [0,1] normierte Sensorwert, die Variable *AvgV* das gleitende Mittel von *Value*.

class *t_SensorGroup*

Datei: `t_sensor.h`

Zweck: faßt gleiche Sensoren zu einer Gruppe zusammen, um zusätzliche Informationnen zu gewinnen

externe Methoden:

`void Init(int n, double avg, int alen, double loc[]);`

initialisiert die gesamte Gruppe von *n* Sensoren.

`void Update(double val[]);`

Die real gemessenen Werte aus dem Array *val* werden in die einzelnen Sensoren eingespeist.

`double GetRange(void);`

gibt die Differenz des größten und kleinsten Sensorwertes im aktuellen Zeitschritt zurück

`double GetBounds(void);`

gibt die Differenz des größten und kleinsten jemals aufgetretenen Sensorwertes zurück

`double Information(void);`

gibt die Information nach SHANNON [SW76] über die Sensorwerte des aktuellen Zeitschrittes zurück

class *t_robot_basic*

Datei: `k_basic.h`

Zweck: Basisobjekt des Roboters. Diese Struktur dient hauptsächlich dazu, die internen Statusvariablen im Controller des *khepera*-Roboters im Hostrechner abzubilden. Die Methoden dieser Objektklasse sind zum Teil Funktionen, die "hardwareseitig-im Roboter implementiert sind. Zum anderen sind es aus mehreren einzelnen Funktionen kombinierte Methoden oder Algorithmen zur Vorverarbeitung und Aufbereitung der gelieferten Daten.

externe Methoden:

`void Init(char *p);`

stellt die Verbindung zum Roboter her und initialisiert alle Variablen auf Null. Der Übergabewert ist eine Zeichenkette und wird als Dateiname einer Gerätedatei interpretiert. Die Zeichenkette muß mit Ziffern enden. Falls ein Leerzeichen in der Zeichenkette enthalten ist, wird der erste Teil der Zeichenkette als Hostname und der zweite Teil als Portnummer interpretiert. In diesem Fall wird versucht, anstelle einer Gerätedatei (serielle Schnittstelle) einen TCP-Socket zu öffnen. Damit besteht die Möglichkeit, auch mit einem Robotersimulator zu arbeiten.

`virtual void SetMotorSpeed(double left, double right);`

setzt die Geschwindigkeit des linken und rechten Rades. Die Eingabewerte werden typischerweise zwischen $-1,0$ und $1,0$ gewählt und beziehen sich auf Vielfache der vorkonfigurierten Maximalgeschwindigkeit von 48 mm/s.

`double GetMotorSpeed(int MotorNummer);`

liest die Motorgeschwindigkeit. Motornummer ist 0 für links und 1 für rechts

`void ReadWheelCounter(int *left, int *right);`

liest die internen Radzählwerke aus

`void SetWheelCounter(int left, int right);`

setzt die internen Radzähler für das linke und rechte Rad auf die vorgegebenen Werte

`ReachWheelCounter(int left, int right);`

läßt die Räder des Roboters so lange drehen, bis der angegebene Zählerstand für die Radzähler erreicht wird

`void SetMotorSpeed(int left, int right);`

setzt die Motorgeschwindigkeit in ganzzahligen Einheiten. Der Wert 6

entspricht der vorkonfigurierten Maximalgeschwindigkeit vom 48 mm/s.

`void GetMotorSpeed(int &L, int &R);`

liest die Radgeschwindigkeit in ganzzahligen Einheiten

`void ReadLEDs(int sensor_class);`

liest den Status der LEDs (an/aus)

`void LED_Ctrl(int led_num, LED_Behavior);`

schaltet die LEDs an, aus oder um

`int GetMotionCtrlFlags(void);`

gibt die Statusvariablen des Bewegungskontrollers im Roboter zurück. Für jeden Radzähler gibt es die drei Zustände „am Ziel“, „bewegt durch Geschwindigkeitsmodus“ und „Positionierfehler“. Die entsprechenden Bitmasken kann man der Headerdatei entnehmen.

`void SetSpeedProfile(int l_max, int r_max, int l_acc, int r_acc);`

Mit dieser Funktion kann der roboterinterne Positionierungskontroller konfiguriert werden. Die Werte sind die zulässigen Maximalwerte für Geschwindigkeit und Beschleunigung pro Rad. Der Roboter benutzt dieses Profil, um die Motorgeschwindigkeit während der Ausführung der Funktion *ReachWheelCounter(..)* zu regeln.

`virtual void Move(double millimeter);`

bewegt den Roboter vorwärts. Dazu wird die Funktion *ReachWheelCounter(..)* benutzt.

`virtual void Turn(double degrees);`

dreht den Roboter. Dazu wird die Funktion *ReachWheelCounter(..)* benutzt. Positive Werte drehen entgegen dem Uhrzeigersinn.

`void UpdateSensors(void);`

Diese Funktion liest nacheinander alle Infrarot- Entfernungssensoren und danach alle Umgebungslichtsensoren und speichert die Werte in einem Array ab. Mit der Funktion *ReadSensor(..)* können die einzelnen Werte abgefragt werden.

`char DistSensor(int i);`

gibt den Wert *false* zurück, wenn der *i*-te Sensor ein simulierter Sensor ist. Bei einem real vorhandenem Sensor wird *true* zurückgegeben.

`virtual void Stop(void);`

setzt die Radgeschwindigkeiten auf Null

`void SetArmPos(int degree);`

dreht den Greifarm in die angegebene Position. Gültige Werte sind -30 bis 210 Grad. Ein Wert von 0 Grad bedeutet: Arm waagerecht nach vorn.

`int ReadGripOhmMeter(void);`

liest den ohmschen Widerstand des im Greifer befindlichen Objektes. Das Ohmmeter ist nicht sehr genau. Der Wert 255 entspricht dem eines Null-Ohm-Widerstandes, der Wert 50 entspricht ungefähr 450 kOhm. Eine Kennlinie kann man aus [SA95] entnehmen.

`char CatchedObject(void);`

gibt den Wert *true* zurück, wenn die Lichtschranke im Greifer unterbrochen ist

`int GrippedWidth(void);`

gibt an, wie weit die Greifzange geschlossen ist

`void Grip(void);`

schließt die Greifzange

`void Release(void);`

öffnet die Greifzange

`virtual void Lift(void);`

hebt den Greifarm und stellt ihn auf ca. 78 Grad ein

`int GetArmPos(void);`

gibt die Greiferposition (Breite des gegriffenen Objektes) zurück

class *t_khepera_msgs*

Datei: `t_msgtype.h`

Zweck: Nachrichtenobjekt, sorgt für effektive Kommunikation zwischen Algorithmus und Roboterhardware. Die Methoden mit „`send_`“- und „`receive_`“-Prefix bilden die passenden Gegenstücke zu den Basismethoden des Roboterobjektes, und werden deshalb nur kurz aufgelistet. Um die neue Schnittstelle zu nutzen, kodiert man

`Robot.MsgQueue.request_motorspeed()` anstelle von

`Robot.GetMotorSpeed()`.

externe Methoden:

```
void send_wheelposition(long l, long r);
void send_positioninfo(long l, long r);
void send_motorspeed(int l, int r);
void send_led(int num, int action);
void send_gripopen(void);
void send_gripclose(void);
void send_armpos(int n);
void request_proximity(void);
void request_ambient(void);
void request_wheelposition(void);
void request_motorspeed(void);
void request_status(void);
void request_gripwidth(void);
void request_armpos(void);
void request_resistor(void);
void request_contact(void);
```

`void request_coords(void);`

führt zum einmaligen Auslesen der Radzählerstände im folgenden Zeitschritt und signalisiert dem Roboter, gleichzeitig eine neue Positionsbestimmung durchzuführen.

`int receive_packet(void);`

leert den Empfangspuffer der seriellen Schnittstelle, parst die empfangenen Zeichenketten und führt Updates der internen Statusvariablen durch. Zurückgegeben wird die zum Empfang benötigte Zeit in Millisekunden, als wären alle Zeichen kurz hintereinander eingetroffen. Es wird nur die Baudrate und die Anzahl der empfangenen Zeichen berücksichtigt.

`int transfer_packet(void);`

sendet alle bisher angesammelten Anfragen und Befehle über die serielle Schnittstelle. Bei doppelten Befehlen wird nur der letzte Befehl gesendet. Zurückgegeben wird die zum Senden benötigte Zeit in Millisekunden.

class *t_robot_ext : public t_robot_basic*

Datei: `k_extend.h`

Zweck: erweitertes Roboterobjekt mit eigenen Variablen für die Welt (t_world), den abgelaufenen Weg (t_PATH), die aktuelle Position (t_location) und der Instanz des Kommunikationsobjektes (t_khepera_msgs)

externe Methoden:

`virtual void SetMotorSpeed(double L, double R);`

überladene Variante der gleichnamigen Basisfunktion, um die Nutzung des Kommunikationsobjektes zu erzwingen.

`void SolveCoords(char queue);`

Neuberechnung der aktuellen Position(t_location) durch die Anwendung der Odometrie-Formeln. Wenn `queue` den Wert *true* hat, wird das Kommunikationsobjekt benutzt, ansonsten wird mit blockierender Ein- /Ausgabe gearbeitet.

`double Kompass(int typ, char rev, double &wid, double &rel);`

Gibt die ungefähre Richtung zur stärksten sichtbaren Lichtquelle in Grad an. Mit der Variable `typ` wählt man die Entfernungs- oder Umgebungslichtsensoren. Wenn `rev` den Wert *true* hat, wird die Richtung der dunkelsten Stelle zurückgegeben. In den Variablenparametern `wid` und `rel` werden Schätzungen zur Streuung und Zuverlässigkeit zurückgegeben.

Literaturverzeichnis

[Can39] W.B. Cannon. *The wisdom of the body.* Norton, New York, 1939.

[DH95] Ralf Der and Michael Herrmann. Self-adjusting reinforcement learning. 1995.

[DP99] Ralf Der and Thomas Pantzer. Emergent robot behavior from the principle of homeokinesis. In A. Löffler, F. Mondada, and U. Rückert, editors, *Proceedings of the 1st International Khepera Workshop*, Experiments with the Mini-Robot Khepera. Heinz-Nixdorf-Institut Verlagsschriftenreihe, 1999.

[DSP99] Ralf Der, Ulrich Steinmetz, and Frank Pasemann. Homeokinesis – a new principle to back up evolution with learning. In M. Mohammadian, editor, *Computational Intelligence for Modelling, Control and Automation*, volume 55 of *Concurrent Systems Engeneering Series*, pages 43–47. IOS Press, 1999.

[EEF90] Werner Ebeling, Andreas Engel, and Rainer Feistel. *Physik der Evolutionsprozesse.* Akademie Verlag, Berlin, 1990.

[Flo99a] Dario Floreano. Evolutionary robotics in behavior engineering and artificial life. 1999.

[Flo99b] Dario Floreano. Reducing human design and increasing adaptibility in evolutionary robotics. 1999.

[GHH+98] Jens Steffen Gutmann, Wolfgang Hatzack, Immanuel Herrmann, Bernhard Nebel, Frank Rittinger, Augustinus Topor, Thilo Weigel, and Bruno Welsch. Reliable self-localization, multirobot sensor integration and basic soccer skills. 1998.

[Gut96] Jens Steffen Gutmann. Vergleich von Algorithmen zur Selbstlokalisierung eines mobilen Roboters. Master's thesis, Universität Ulm, 1996.

[HPG97] Michael Herrmann, Klaus Pawelzik, and Theo Geisel. Self-localization by hidden representations. 1997.

[KS97] Sven König and Reid G. Simmons. Self-localization by hidden representations. 1997.

[LPJ+96] L.Yriarte, P.Deplanques, J.Sallantin, P.Reitz, R.Zapata, B.Burg, and F.Arlabosse. An architecture for modelling and validation. Application to mobile robotics. 1996.

[May90] P. S. Maybeck. The kalman filter: An introduction to concepts. In I.J. Cox and G.T. Wilfong, editors, *Autonomous Robot Vehicles*, pages 194–204. Springer Verlag, 1990.

[MLP+98] Ralf Möller, Dimitrios Lambrinos, Rolf Pfeifer, Thomas Labhart, and Rüdiger Wehner. Modelling ant navigation with an autonomous agent. In Rolf Pfeifer, Bruce Blumberg, Jean-Arcady Meyer, and Steward W. Wilson, editors, *Proc. of the Fifth International Conference on Simulation of Adaptive Behavior*, volume 5 of *From Animals to Animats*, 1998.

[MM80] M. L. Mittelstaedt and H. Mittelstaedt. Homing by path integration in a mammal. *Naturwissenschaften*, pages 566–567, 1980.

[NF97] Stefano Nolfi and Dario Floreano. God save the red queen. In *Proceedings of the Second Annual Conference*, pages 398–406, 1997.

[NML95] Stefano Nolfi, Orazio Miglino, and Henrik Hautop Lund. Evolving mobile robots in simulated and real environments. 1995.

[PS98] Rolf Pfeifer and Christian Scheier. Exploiting embodiment for category learning. In Rolf Pfeifer, Bruce Blumberg, Jean-Arcady Meyer, and Steward W. Wilson, editors, *Proc. of the Fifth International Conference on Simulation of Adaptive Behavior*, volume 5 of *From Animals to Animats*, 1998.

[SA95] K-Team SA. Khepera gripper user manual. Technical report, 1995.

[SG98] Erol Sahin and Paolo Gaudiano. Visual looming as a range sensor for mobile robots. In Rolf Pfeifer, Bruce Blumberg, Jean-Arcady Meyer, and Steward W. Wilson, editors, *Proc. of the Fifth International Conference on*

Simulation of Adaptive Behavior, volume 5 of *From Animals to Animats*, 1998.

[SG99] Erol Sahin and Paolo Gaudiano. Kite: The khepera integrated testing environment. In A. Löffler, F. Mondada, and U. Rückert, editors, *Proceedings of the 1st International Khepera Workshop*, Experiments with the Mini-Robot Khepera. Heinz-Nixdorf-Institut Verlagsschriftenreihe, 1999.

[SW76] Claude E. Shannon and Warren Weaver. *Mathematische Grundlagen der Informationstheorie*. R. Oldenbourg, München, 1976.

[Wil71] E.O. Wilson. *The Insect Societies*. Harvard University Press, Cambridge, Mass., 1971.

Erklärung

Hiermit versichere ich, daß ich die vorliegende Arbeit selbst angefertigt und nur die angegebenen Hilfsmittel benutzt habe.

Datum Unterschrift